Information technology — Database languages — SQL —

Part 2: Foundation (SQL/Foundation)

1 Scope

This part of ISO/IEC 9075 defines the data structures and basic operations on SQL-data. It provides functional capabilities for creating, accessing, maintaining, controlling, and protecting SQL-data.

This part of ISO/IEC 9075 specifies the syntax and semantics of a database language:

— For specifying and modifying the structure and the integrity constraints of SQL-data.

— For declaring and invoking operations on SQL-data and cursors.

— For declaring database language procedures.

— For embedding SQL-statements in a compilation unit that otherwise conforms to the standard for a particular programming language (host language).

— For deriving an equivalent compilation unit that conforms to the particular programming language standard. In that equivalent compilation unit, each embedded SQL-statement has been replaced by one or more statements in the host language, some of which invoke an SQL externally-invoked procedure that, when executed, has an effect equivalent to executing the SQL-statement.

— For direct invocation of SQL-statements.

— To support dynamic preparation and execution of SQL-statements.

This part of ISO/IEC 9075 provides a vehicle for portability of data definitions and compilation units between SQL-implementations.

This part of ISO/IEC 9075 provides a vehicle for interconnection of SQL-implementations.

Implementations of this part of ISO/IEC 9075 may exist in environments that also support application programming languages, end-user query languages, report generator systems, data dictionary systems, program library systems, and distributed communication systems, as well as various tools for database design, data administration, and performance optimization.

This page intentionally left blank.

2 Normative references

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

2.1 JTC1 standards

[ISO646] ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange.*

[ISO1539] ISO/IEC 1539-1:1997, *Information technology — Programming languages — Fortran — Part 1: Base language.*

ISO/IEC 1539-1:1997/Cor.1:2001.

ISO/IEC 1539-1:1997/Cor.2:2002.

[ISO1989] ISO 1989:1985, *Programming languages — COBOL.* (Endorsement of ANSI X3.23-1985).

ISO/IEC 1989:1985/Amd.1:1992, *Intrinsic function module*

ISO/IEC 1989:1985/Amd.2:1994, *Correction and clarification amendment for COBOL*

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

[ISO6429] ISO/IEC 6429:1992, *Information technology — Control functions for coded character sets*

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

[ISO8601] ISO 8601:2000, *Data elements and interchange formats — Information interchange — Representation of dates and times.*

[ISO8649] ISO/IEC 8649:1996, *Information technology — Open Systems Interconnection — Service Definition for the Association Control Service Element.*

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

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

[Latin1] ISO/IEC 8859-1:1998, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*

[Framework] ISO/IEC 9075-1:2003, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework).*

[Schemata] ISO/IEC 9075-11:2003, *Information technology — Database languages — SQL — Part 11: Information and Definition Schemas (SQL/Schemata).*

[ISO9579] ISO/IEC 9579:2000, *Information technology — Remote database access for SQL with security enhancement.*

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

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

[ISO10026] ISO/IEC 10026-2:1998, *Information technology — Open Systems Interconnection — Distributed Transaction Processing — Part 2: OSI TP Service.*

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

[UCS] ISO/IEC 10646-1:2000, *Information technology — Universal Multi-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

[UCSsupp] ISO/IEC 10646-2:2001, *Information technology — Universal Multi-Octet Coded Character Set (UCS) — Part 2: Supplementary Planes.*

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

[ISO14651] ISO/IEC 14651:2001, *Information technology — International string ordering and comparison — Method for comparing character strings and description of the common template tailorable ordering.*

2.2 Other international standards

[Unicode 3.0] The Unicode Consortium, *The Unicode Standard, Version 3.0*, Reading, MA, Addison-Wesley Developers Press, 2000. ISBN 0-201-61633-5.

[Unicode 3.1] The Unicode Consortium, *The Unicode Standard, Version 3.1.0, Unicode Standard Annex #27: Unicode 3.1 (which amends The Unicode Standard, Version 3.0).* 2001-03-23.
`http://www.unicode.org/unicode/reports/tr27/`

[Unicode10] Davis, Mark and Whistler, Ken. *Unicode Technical Standard #10, Unicode Collation Algorithm, Version 8.0*, 2001-03-23. The Unicode Consortium.
`http://www.unicode.org/unicode/reports/tr10/tr10-8.html`

[Unicode15] Davis, Mark and Dürst, Martin. *Unicode Standard Annex #15, Unicode Normalization Forms, Version 21.0*, 2001-03-23. The Unicode Consortium.
`http://www.unicode.org/unicode/reports/tr15/tr15-21.html`

[Unicode19] Davis, Mark. *Unicode Standard Annex #19, UTF-32, Version 8.0*, 2001-03-23. The Unicode Consortium.
`http://www.unicode.org/unicode/reports/tr19/tr19-8.html`

[IANA] The Internet Assigned Numbers Authority, *Character sets*
`http://www.iana.org/assignments/character-sets`

3 Definitions, notations, and conventions

3.1 Definitions

3.1.1 Definitions taken from ISO/IEC 10646

For the purposes of this part of ISO/IEC 9075, the definitions of the following terms given in ISO/IEC 10646 apply:

3.1.1.1 character

NOTE 1 — This is identical to the Unicode definition of *abstract character*. In ISO/IEC 9075, when the relevant character repertoire is UCS, a character can be thought of as *that which is represented by one code point.*

3.1.1.2 repertoire

3.1.2 Definitions taken from ISO/IEC 14651

For the purposes of this part of ISO/IEC 9075, the definitions of the following terms given in ISO/IEC 14651 apply:

3.1.2.1 collation

3.1.3 Definitions taken from Unicode

For the purposes of this part of ISO/IEC 9075, the definitions of the following terms given in The Unicode Standard apply:

3.1.3.1 character encoding form

3.1.3.2 code point

3.1.3.3 code unit

3.1.3.4 control character

3.1.3.5 noncharacter

3.1.3.6 normalization

3.1.3.7 transcoding

3.1.4 Definitions taken from ISO 8601

For the purposes of this part of ISO/IEC 9075, the definitions of the following terms given in ISO 8601 apply:

3.1.4.1 Coordinated Universal Time (UTC)

3.1.4.2 date (date, calendar in ISO 8601)

3.1.5 Definitions taken from Part 1

For the purposes of this part of ISO/IEC 9075, the definitions given in ISO/IEC 9075-1 apply.

3.1.6 Definitions provided in Part 2

For the purposes of this part of ISO/IEC 9075, in addition to those definitions taken from other sources, the following definitions apply:

3.1.6.1 assignable (of types, taken pairwise): The characteristic of a data type *T1* that permits a value of *T1* to be assigned to a site of a specified data type *T2* (where *T1* and *T2* may be the same data type.

3.1.6.2 assignment: The operation that causes the value at a site *T* (known as the *target*) to be a given value *S* (known as the *source*). Assignment is frequently indicated by the use of the phrase "*T* is set to *S*" or "the value of *T* is set to *S*".

3.1.6.3 attribute: A component of a structured type. Each value *V* in structured type *T* has exactly one attribute value for each attribute *A* of *T*. The characteristics of an attribute are specified by an attribute descriptor. The value of an attribute may be retrieved as the result of the invocation *A*(*V*) of the observer function for that attribute.

3.1.6.4 cardinality (of a collection): The number of elements in that collection. Those elements need not necessarily have distinct values. The objects to which this concept applies includes tables and the values of collection types.

3.1.6.5 comparable (of a pair of values): Capable of being compared, according to the rules of Subclause 8.2, "<comparison predicate>". In most, but not all, cases, the values of a data type can be compared one with another. For the specification of comparability of individual data types, see Subclause 4.2, "Character strings", through Subclause 4.10, "Collection types".

3.1.6.6 constructor function: A niladic SQL-invoked function of which exactly one is implicitly specified for every structured type. An invocation of the constructor function for data type *T* returns a value *V* of the most specific type *T* such that *V* is not null and, for every observer function *O* defined for *T*, the invocation *O*(*V*) returns the default value of the attribute corresponding to *O*.

3.1.6.7 declared type (of an expression denoting a value or anything that can be referenced to denote a value, such as, for example, a parameter, column, or variable): The unique data type that is common to every value that might result from evaluation of that expression.

3.1.6.8 distinct (of a pair of comparable values): Capable of being distinguished within a given context. Informally, not equal, not both null. A null value and a non-null value are distinct.

For two non-null values, the following rules apply:

— Two values of predefined type or reference type are distinct if and only if they are not equal.

— If two values *V1* and *V2* are of a user-defined type whose comparison form is RELATIVE or MAP and the result of comparing them for equality according to Subclause 8.2, "<comparison predicate>", is *Unknown*, then it is implementation-dependent whether they are distinct or not; otherwise, they are distinct if and only if they are not equal.

— If two values *V1* and *V2* are of a user-defined type whose comparison form is STATE, then they are distinct if their most specific types are different, or if there is an attribute *A* of their common most specific type such that the value of *A* in *V1* and the value of *A* in *V2* are distinct.

— Two rows are distinct if and only if at least one of their pairs of respective fields is distinct.

— Two arrays that do not have the same cardinality are distinct.

— Two arrays that have the same cardinality and in which there exists at least one ordinal position *P* such that the array element at position *P* in one array is distinct from the array element at position *P* in the other array are distinct.

— Two multisets *A* and *B* are distinct if there exists a value *V* in the element type of *A* or *B*, including the null value, such that the number of elements in *A* that are not distinct from *V* does not equal the number of elements in *B* that are not distinct from *V*.

NOTE 2 — The result of evaluating whether or not two comparable values are distinct is never *Unknown*. The result of evaluating whether or not two values that are not comparable (for example, values of a user-defined type that has no comparison type) are distinct is not defined.

3.1.6.9 duplicates: Two or more members of a multiset that are not distinct.

3.1.6.10 dyadic (of operators, functions, and procedures): Having exactly two operands or parameters.

NOTE 3 — An example of a dyadic operator in this part of ISO/IEC 9075 is "–", specifying the subtraction of the right operand from the left operand. An example of a dyadic function is POSITION.

3.1.6.11 element type (of a collection type and every value in that collection type): The declared type specified in the definition of a collection type *CT* that is common to every element of every value of type *CT*.

3.1.6.12 equal (of a pair of comparable values): Yielding *True* if passed as arguments in a <comparison predicate> in which the <comp op> is <equals operator>. (see Subclause 8.2, "<comparison predicate>").

3.1.6.13 external routine: An SQL-invoked routine whose routine body is an external body reference that identifies a program written in a standard programming language other than SQL.

3.1.6.14 fixed-length: A characteristic of character strings that restricts a string to contain exactly one number of characters, known as the length in characters of the string.

3.1.6.15 identical (of a pair of comparable values): Indistinguishable, in the sense that it is impossible, by any means specified in ISO/IEC 9075, to detect any difference between them. For the full definition, see Subclause 9.8, "Determination of identical values".

3.1.6.16 interface (of a structured type): The set comprising every function such that the declared type of at least one of its parameters or result is that structured type.

3.1.6.17 monadic (of operators, functions, and procedures): Having exactly one operand or parameter.

NOTE 4 — An example of a monadic arithmetic operator in this part of ISO/IEC 9075 is "–", specifying the negation of the operand. An example of a monadic function is CHARACTER_LENGTH, specifying the length in characters of the argument.

3.1.6.18 most specific type (of a value): The unique data type of which every data type of that value is a supertype.

3.1.6.19 mutator function: A dyadic, type-preserving function *M* whose definition is implied by the definition of some attribute *A* (of declared type *AT*) of some user-defined type *T*. The first parameter of *M* is a result SQL parameter of declared type *T*, which is also the result type of *M*. The second parameter of *M* is of declared type *AT*. If *V* is some value in *T* and *AV* is some value in *AT*, then the invocation *M*(*V*, *AV*) returns the value *V1* such that *V1* differs from *V* only in its value for attribute *A*, if at all. The most specific type of *V1* is the most specific type of *V*.

3.1.6.20 n-adic operator: An operator having a variable number of operands (informally: *n* operands).

NOTE 5 — An example of an *n*-adic operator in this part of ISO/IEC 9075 is COALESCE.

3.1.6.21 niladic (of functions and procedures): Having no parameters.

3.1.6.22 observer function: An SQL-invoked function *M* implicitly defined by the definition of attribute *A* of a structured type *T*. If *V* is some value in *T* and the declared type of *A* is *AT*, then the invocation of *M*(*V*) returns some value *AV* in *AT*. *AV* is then said to be the value of attribute *A* in *V*.

3.1.6.23 redundant duplicates: All except one of any collection of duplicate values or rows.

3.1.6.24 REF value: A value that references some site.

3.1.6.25 reference type: A data type all of whose values are potential references to sites of one specified data type.

3.1.6.26 referenced type: The declared type of the values at sites referenced by values of a particular reference type.

3.1.6.27 referenced value: The value at the site referenced by a REF value.

3.1.6.28 result SQL parameter: An SQL parameter that specifies RESULT.

3.1.6.29 result data type: The declared type of the result of an SQL-invoked function.

3.1.6.30 signature (of an SQL-invoked routine): The name of an SQL-invoked routine, the position and declared type of each of its SQL parameters, and an indication of whether it is an SQL-invoked function or an SQL-invoked procedure.

3.1.6.31 SQL argument: An expression denoting a value to be substituted for an SQL parameter in an invocation of an SQL-invoked routine.

3.1.6.32 SQL-invoked routine: A routine that is allowed to be invoked only from within SQL.

3.1.6.33 SQL parameter: A parameter declared as part of the signature of an SQL-invoked routine.

3.1.6.34 SQL routine: An SQL-invoked routine whose routine body is written in SQL.

3.1.6.35 **subfield (of a row type):** A field that is a field of a row type *RT* or a field of a row type *RT2* that is the declared type of a field that is a subfield of *RT*.

3.1.6.36 **subtype (of a data type):** A data type *T2* such that every value of *T2* is also a value of data type *T1*. If *T1* and *T2* are not compatible, then *T2* is a *proper subtype* of *T1*. "Compatible" is defined in Subclause 4.1, "Data types". See also **supertype**.

3.1.6.37 **supertype (of a data type):** A data type *T1* such that every value of *T2* is also a value of data type *T1*. If *T1* and *T2* are not compatible, then *T1* is a *proper supertype* of *T2*. "Compatible" is defined in Subclause 4.1, "Data types". See also **subtype**.

3.1.6.38 **transliteration:** A method of translating characters in one character set into characters of the same or a different character set.

3.1.6.39 **type-preserving function:** An SQL-invoked function, one of whose parameters is a result SQL parameter. The most specific type of the value returned by an invocation of a type-preserving function is identical to the most specific type of the SQL argument value substituted for the result SQL parameter.

3.1.6.40 **user-defined type:** A type whose characteristics are specified by a user-defined type descriptor.

3.1.6.41 **variable-length:** A characteristic of character strings and binary strings that allows a string to contain any number of characters or octets, respectively, between 0 (zero) and some maximum number, known as the maximum length in characters or octets, respectively, of the string.

3.1.6.42 **white space:** Characters used to separate tokens in SQL text; white space may be required (for example, to separate <nondelimiter token>s from one another) and may be used between any two tokens for which there are no rules prohibiting such use.

White space is any character in the Unicode General Category classes "Zs", "Zl", and "Zp", or any of the following characters:

— U+0009, Horizontal Tabulation

— U+000A, Line Feed

— U+000B, Vertical Tabulation

— U+000C, Form Feed

— U+000D, Carriage Return

— U+0085, Next Line

NOTE 6 — The normative provisions of this International Standard impose no requirement that any character set have equivalents for any of these characters except U+0020 (<space>); however, by reference to this definition of white space, they do impose the requirement that every equivalent for one of these shall be recognized as a white space character.

NOTE 7 — The Unicode General Category classes "Zs", "Zl", and "Zp" are assigned to Unicode characters that are, respectively, space separators, line separators, and paragraph separators.

The only character that is a member of the Unicode General Category class "Zl" is U+2028, Line Separator. The only character that is a member of the Unicode General Category class "Zp" is U+2029, Paragraph Separator. The characters that are members of the Unicode General Category class "Zs" are: U+0020, Space, U+00A0, No-Break Space, U+1680, Ogham Space Mark, U+2000, En Quad, U+2001, Em Quad, U+2002, En Space, U+2003, Em Space, U+2004, Three-Per-Em Space, U+2005, Four-Per-Em Space, U+2006, Six-Per-Em Space, U+2007, Figure Space, U+2008, Punctuation Space, U+2009, Thin Space, U+200A, Hair Space, U+202F, Narrow No-Break Space, and U+3000, Ideographic Space.

3.2 Notation

The notation used in this part of ISO/IEC 9075 is defined in ISO/IEC 9075-1.

3.3 Conventions

The conventions used in this part of ISO/IEC 9075 are defined in ISO/IEC 9075-1, with the following additions.

3.3.1 Use of terms

3.3.1.1 Other terms

An SQL-statement *S1* is said to be executed as a *direct result of executing an SQL-statement* if *S1* is the SQL-statement contained in an <externally-invoked procedure> or <SQL-invoked routine> that has been executed.

An SQL-statement *S1* is said to be executed as a *direct result of executing an SQL-statement* if *S1* is the value of an <SQL statement variable> referenced by an <execute immediate statement> contained in an <externally-invoked procedure> that has been executed, or if *S1* was the value of the <SQL statement variable> that was associated with an <SQL statement name> by a <prepare statement> and that same <SQL statement name> is referenced by an <execute statement> contained in an <externally-invoked procedure> that has been executed.

4 Concepts

4.1 Data types

4.1.1 General introduction to data types

A data type is a set of representable values. Every representable value belongs to at least one data type and some belong to several data types.

Exactly one of the data types of a value *V*, namely the most specific type of *V*, is a subtype of every data type of *V*. A <value expression> *E* has exactly one declared type, common to every possible result of evaluating *E*. Items that can be referenced by name, such as SQL parameters, columns, fields, attributes, and variables, also have declared types.

SQL supports three sorts of data types: *predefined data types*, *constructed types*, and *user-defined types*. Predefined data types are sometimes called "built-in data types", though not in this International Standard. User-defined types can be defined by a standard, by an implementation, or by an application.

A constructed type is specified using one of SQL's data type constructors, ARRAY, MULTISET, REF, and ROW. A constructed type is either an array type, a multiset type, a reference type, or a row type, according to whether it is specified with ARRAY, MULTISET, REF, or ROW, respectively. Array types and multiset types are known generically as *collection types*.

Every predefined data type is a subtype of itself and of no other data types. It follows that every predefined data type is a supertype of itself and of no other data types. The predefined data types are individually described in each of Subclause 4.2, "Character strings", through Subclause 4.6, "Datetimes and intervals".

Row types, reference types and collection types are described in Subclause 4.8, "Row types", Subclause 4.9, "Reference types", Subclause 4.10, "Collection types", respectively.

4.1.2 Naming of predefined types

SQL defines predefined data types named by the following <key word>s: CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT, BINARY LARGE OBJECT, NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DOUBLE PRECISION, BOOLEAN, DATE, TIME, TIMESTAMP, and INTERVAL. These names are used in the *type designators* that constitute the *type precedence lists* specified in Subclause 9.5, "Type precedence list determination".

For reference purposes:

— The data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are collectively referred to as *character string types*.

— The data type BINARY LARGE OBJECT is referred to as the *binary string type* and the values of binary string types are referred to as *binary strings*.

— The data types CHARACTER LARGE OBJECT and BINARY LARGE OBJECT are collectively referred to as *large object string types* and the values of large object string types are referred to as *large object strings*.

— Character string types and binary string types are collectively referred to as *string types* and values of string types are referred to as *strings*.

— The data types NUMERIC, DECIMAL, SMALLINT, INTEGER, and BIGINT are collectively referred to as *exact numeric types*.

— The data types FLOAT, REAL, and DOUBLE PRECISION are collectively referred to as *approximate numeric types*.

— Exact numeric types and approximate numeric types are collectively referred to as *numeric types*. Values of numeric types are referred to as *numbers*.

— The data types TIME WITHOUT TIME ZONE and TIME WITH TIME ZONE are collectively referred to as *time types* (or, for emphasis, as time with or without time zone).

— The data types TIMESTAMP WITHOUT TIME ZONE and TIMESTAMP WITH TIME ZONE are collectively referred to as *timestamp types* (or, for emphasis, as timestamp with or without time zone).

— The data types DATE, TIME, and TIMESTAMP are collectively referred to as *datetime types*.

— Values of datetime types are referred to as *datetimes*.

— The data type INTERVAL is referred to as an *interval type*. Values of interval types are called *intervals*.

Each data type has an associated data type descriptor; the contents of a data type descriptor are determined by the specific data type that it describes. A data type descriptor includes an identification of the data type and all information needed to characterize a value of that data type.

Subclause 6.1, "<data type>", describes the semantic properties of each data type.

4.1.3 Non-predefined and non-SQL types

A structured type *ST* is *directly based on* a data type *DT* if any of the following are true:

— *DT* is the declared type of some attribute of *ST*.

— *DT* is a direct supertype of *ST*.

— *DT* is a direct subtype of *ST*.

— *DT* is compatible with *ST*.

A collection type *CT* is *directly based on* a data type *DT* if *DT* is the element type of *CT*.

A row type *RT* is *directly based on* a data type *DT* if *DT* is the declared type of some field (or the data type of the domain of some field) whose descriptor is included in the descriptor of *RT*.

A data type *DT1* is *based on* a data type *DT2* if *DT1* is compatible with *DT2*, *DT1* is directly based on *DT2*, or *DT1* is directly based on some data type that is based on *DT2*.

A type *TY* is *usage-dependent* on a user-defined type *UDT* if one of the following conditions is true:

— *TY* is *UDT*.

— *TY* is a reference type whose referenced type is *UDT*.

— *TY* is a row type, and the declared type of a field of *TY* is usage-dependent on *UDT*.

— *TY* is a collection type, and the declared element type of *TY* is usage-dependent on *UDT*.

Each host language has its own data types, which are separate and distinct from SQL data types, even though similar names may be used to describe the data types. Mappings of SQL data types to data types in host languages are described in Subclause 11.50, "<SQL-invoked routine>", and Subclause 20.1, "<embedded SQL host program>". Not every SQL data type has a corresponding data type in every host language.

4.1.4 Comparison and ordering

Ordering and comparison of values of the predefined data types requires knowledge only about those predefined data types. However, to be able to compare and order values of constructed types or of user-defined types, additional information is required. We say that some type *T* is *S*-ordered, for some set of types *S*, if, in order to compare and order values of type *T*, information about ordering at least one of the types in *S* is first required. A definition of *S*-ordered is required for several *S* (that is, for several sets of types), but not for all possible such sets.

The general definition of *S*-ordered is this:

Let *T* be a type and let *S* be a set of types. *T* is *S-ordered* if one of the following is true:

— *T* is a member of *S*.

— *T* is a row type and the declared type of some field of *T* is *S*-ordered.

— *T* is a collection type and the element type of *T* is *S*-ordered.

— *T* is a structured type whose comparison form is STATE and the declared type of some attribute of *T* is *S*-ordered.

— *T* is a user-defined type whose comparison form is MAP and the return type of the SQL-invoked function that is identified by the <map function specification> is *S*-ordered.

— *T* is a reference type with a derived representation and the declared type of some attribute enumerated by the <derived representation> is *S*-ordered.

The notion of *S*-ordered is applied in the following definitions:

— A type *T* is *LOB-ordered* if *T* is *L*-ordered, where *L* is the set of large object types.

— A type *T* is *array-ordered* if *T* is *ARR*-ordered, where *ARR* is the set of array types.

— A type *T* is *multiset-ordered* if *T* is *MUL*-ordered, where *MUL* is the set of multiset types.

— A type *T* is *reference-ordered* if *T* is *REF*-ordered, where *REF* is the set of reference types.

— A type *T* is *DT-EC-ordered* if *T* is *DTE*-ordered, where *DTE* is the set of distinct types with EQUALS ONLY comparison form (DT-EC stands for "distinct type-equality comparison").

— A type *T* is *DT-FC-ordered* if *T* is *DTF*-ordered, where *DTF* is the set of distinct types with FULL comparison form.

— A type *T* is *DT-NC-ordered* if *T* is *DTN*-ordered, where *DTN* is the set of distinct types with no comparison form.

— A type *T* is *ST-EC-ordered* if *T* is *STE*-ordered, where *STE* is the set of structured types with EQUALS ONLY comparison form.

— A type *T* is *ST-FC-ordered* if *T* is *STF*-ordered, where *STF* is the set of structured types with FULL comparison form.

— A type *T* is *ST-NC-ordered* if *T* is *STN*-ordered, where *STN* is the set of structured types with no comparison form.

— A type *T* is *ST-ordered* if *T* is ST-EC-ordered, ST-FC-ordered, or ST-NC-ordered.

— A type *T* is *UDT-EC-ordered* if *T* is either DT-EC-ordered or ST-EC-ordered (UDT stands for "user-defined type").

— A type *T* is *UDT-FC-ordered* if *T* is either DT-FC-ordered or ST-FC-ordered

— A type *T* is *UDT-NC-ordered* if *T* is either DT-NC-ordered or ST-NC-ordered.

The notion of a *constituent* of a declared type *DT* is defined recursively as follows:

— *DT* is a constituent of *DT*.

— If *DT* is a row type, then the declared type of each field of *DT* is a constituent of *DT*.

— If *DT* is a collection type, then the element type of *DT* is a constituent of *DT*.

— Every constituent of a constituent of *DT* is a constituent of *DT*.

Two data types, *T1* and *T2*, are said to be compatible if *T1* is assignable to *T2*, *T2* is assignable to *T1*, and their descriptors include the same data type name. If they are row types, it shall further be the case that the declared types of their corresponding fields are pairwise compatible. If they are collection types, it shall further be the case that their element types are compatible. If they are reference types, it shall further be the case that their referenced types are compatible.

NOTE 8 — The data types "CHARACTER(*n*) CHARACTER SET *CS1*" and "CHARACTER(*m*) CHARACTER SET *CS2*", where $CS1 \neq CS2$, have descriptors that include the same data type name (CHARACTER), but are not mutually assignable; therefore, they are not compatible.

4.2 Character strings

4.2.1 Introduction to character strings

A character string is a sequence of characters. All the characters in a character string are taken from a single character set. A character string has a length, which is the number of characters in the sequence. The length is 0 (zero) or a positive integer.

A *character string type* is described by a character string type descriptor. A character string type descriptor contains:

— The name of the specific character string type (CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT; NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, and NATIONAL CHARACTER LARGE OBJECT are represented as CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT, respectively).

— The length or maximum length in characters of the character string type.

— The catalog name, schema name, and character set name of the character set of the character string type.

— The catalog name, schema name, and collation name of the collation of the character string type.

A *character large object type* is a character string type where the name of the specific character string type is CHARACTER LARGE OBJECT. A value of a character large object type is a *large object character string*.

The character set of a character string type may be specified explicitly or implicitly.

The <key word>s NATIONAL CHARACTER are used to specify an implementation-defined character set. Special syntax (`N'string'`) is provided for representing literals in that character set.

With two exceptions, a character string expression is assignable only to sites of a character string type whose character set is the same. The exceptions are as specified in Subclause 4.2.8, "Universal character sets", and such other cases as may be implementation-defined. If a store assignment would result in the loss of non-<space> characters due to truncation, then an exception condition is raised. If a retrieval assignment or evaluation of a <cast specification> would result in the loss of characters due to truncation, then a warning condition is raised.

Character sets fall into three categories: those defined by national or international standards, those defined by SQL-implementations, and those defined by applications. The character sets defined by ISO/IEC 10646 and The Unicode Standard are known as *Universal Character Sets* (UCS) and their treatment is described in Subclause 4.2.8, "Universal character sets". Every character set contains the <space> character (equivalent to U+0020). An application defines a character set by assigning a new name to a character set from one of the first two categories. They can be defined to "reside" in any schema chosen by the application. Character sets defined by standards or by SQL-implementations reside in the Information Schema (named INFORMATION_SCHEMA) in each catalog, as do collations defined by standards and collations, transliterations, and transcodings defined by SQL-implementations.

NOTE 9 — The Information Schema is defined in ISO/IEC 9075-11.

4.2.2 Comparison of character strings

Two character strings are comparable if and only if either they have the same character set or there exists at least one collation that is applicable to both their respective character sets.

A *collation* is defined by ISO/IEC 14651 as "a process by which two strings are determined to be in exactly one of the relationships of less than, greater than, or equal to one another". Each collation known in an SQL-environment is applicable to one or more character sets, and for each character set, one or more collations are applicable to it, one of which is associated with it as its *character set collation.*

Anything that has a declared type can, if that type is a character string type, be associated with a collation applicable to its character set; this is known as a *declared type collation.* Every declared type that is a character string type has a collation derivation, this being either *none*, *implicit*, or *explicit.* The collation derivation of a declared type with a declared type collation that is explicitly or implicitly specified by a <data type> is *implicit.* If the collation derivation of a declared type that has a declared type collation is not *implicit*, then it is *explicit.* The collation derivation of an expression of character string type that has no declared type collation is *none.*

An operation that explicitly or implicitly involves character string comparison is a *character comparison operation.* At least one of the operands of a character comparison operation shall have a declared type collation.

There may be an SQL-session collation for some or all of the character sets known to the SQL-implementation (see Subclause 4.37, "SQL-sessions").

The collation used for a particular character comparison is specified by Subclause 9.13, "Collation determination".

The comparison of two character string expressions depends on the collation used for the comparison (see Subclause 9.13, "Collation determination"). When values of unequal length are compared, if the collation for the comparison has the NO PAD characteristic and the shorter value is equal to some prefix of the longer value, then the shorter value is considered less than the longer value. If the collation for the comparison has the PAD SPACE characteristic, for the purposes of the comparison, the shorter value is effectively extended to the length of the longer by concatenation of <space>s on the right.

For every character set, there is at least one collation.

4.2.3 Operations involving character strings

4.2.3.1 Operators that operate on character strings and return character strings

<concatenation operator> is an operator, | |, that returns the character string made by joining its character string operands in the order given.

<character substring function> is a triadic function, SUBSTRING, that returns a string extracted from a given string according to a given numeric starting position and a given numeric length.

<regular expression substring function> is a triadic function, SUBSTRING, distinguished by the keywords SIMILAR and UESCAPE. It has three parameters: a source character string, a pattern string, and an escape character. It returns a result string extracted from the source character string by pattern matching.

— **Step 1**: The escape character is exactly one character in length. As indicated in Figure 1, "Operation of <regular expression substring function>", the escape character precedes two instances of <double quote> that are used to partition the pattern string into three subpatterns (identified as *R1*, *R2*, and *R3*).

— **Step 2**: If the source string *S* does not satisfy the predicate

```
'S' SIMILAR TO 'R1' || 'R2' || 'R3'
```

then the result is the null value.

— **Step 3**: Otherwise, *S* is partitioned into two substrings *S1* and *S23* such that *S1* is the shortest initial substring of *S* such that the following condition is satisfied:

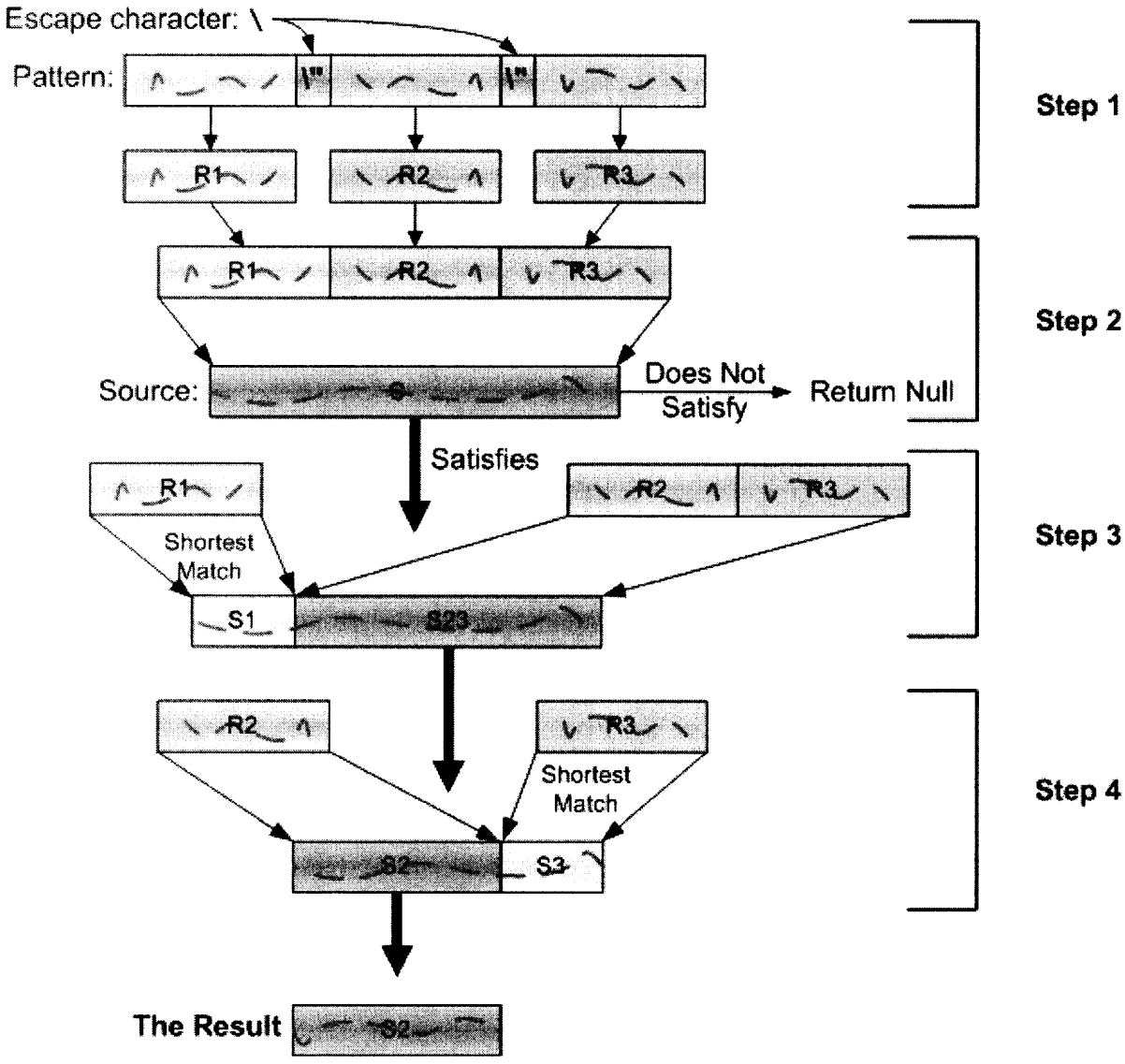
```
'S1' SIMILAR TO 'R1' AND
  'S23' SIMILAR TO '(' || 'R2' || 'R3' || ')'
```

— **Step 4**: Next, *S23* is partitioned into two substrings *S2* and *S3* such that *S3* is the shortest final substring such that the following condition is satisfied:

```
'S2' SIMILAR TO 'R2' AND 'S3' SIMILAR TO 'R3'
```

The result of the <regular expression substring function> is *S2*.

Figure 1 — Operation of <regular expression substring function>



<character overlay function> is a function, OVERLAY, that modifies a string argument by replacing a given substring of the string, which is specified by a given numeric starting position and a given numeric length, with another string (called the replacement string). When the length of the substring is zero, nothing is removed from the original string and the string returned by the function is the result of inserting the replacement string into the original string at the starting position.

<fold> is a pair of functions for converting all the lower case and title case characters in a given string to upper case (UPPER) or all the upper case and title case characters to lower case (LOWER). A lower case character is a character in the Unicode General Category class "Ll" (lower-case letters). An upper case character is a

character in the Unicode General Category class "Lu" (upper-case letters). A title case character is a character in the Unicode General Category class "Lt" (title-case letters).

NOTE 10 — Case correspondences are not always one-to-one: the result of case folding may be of a different length in characters than the source string. For example, U+00DF, "ß", Latin Small Letter Sharp S, becomes "SS" when folded to upper case.

<transcoding> is a function that invokes an installation-supplied transcoding to return a character string *S2* derived from a given character string *S1*. It is intended, though not enforced by this part of ISO/IEC 9075, that *S2* be exactly the same sequence of characters as *S1*, but encoded according to some different character encoding form. A typical use might be to convert a character string from two-octet UCS to one-octet Latin1 or *vice versa*.

<trim function> is a function that returns its first string argument with leading and/or trailing pad characters removed. The second argument indicates whether leading, or trailing, or both leading and trailing pad characters should be removed. The third argument specifies the pad character that is to be removed.

<character transliteration> is a function for changing each character of a given string according to some many-to-one or one-to-one mapping between two not necessarily distinct character sets. The mapping, rather than being specified as part of the function, is some external function identified by a <transliteration name>.

For any pair of character sets, there are zero or more transliterations that may be invoked by a <character transliteration>. A transliteration is described by a transliteration descriptor. A transliteration descriptor includes:

— The name of the transliteration.

— The name of the character set from which it translates.

— The name of the character set to which it translates.

— The specific name of the SQL-invoked routine that performs the transliteration.

4.2.3.2 Other operators involving character strings

<length expression> returns the length of a given character string, as an exact numeric value, in characters or octets according to the choice of function.

<position expression> determines the first position, if any, at which one string, *S1*, occurs within another, *S2*. If *S1* is of length zero, then it occurs at position 1 (one) for any value of *S2*. If *S1* does not occur in *S2*, then zero is returned. The declared type of a <position expression> is exact numeric.

<like predicate> uses the triadic operator LIKE (or the inverse, NOT LIKE), operating on three character strings and returning a Boolean. LIKE determines whether or not a character string "matches" a given "pattern" (also a character string). The characters <percent> and <underscore> have special meaning when they occur in the pattern. The optional third argument is a character string containing exactly one character, known as the "escape character", for use when a <percent>, <underscore>, or the "escape character" itself is required in the pattern without its special meaning.

<similar predicate> uses the triadic operator SIMILAR (or the inverse, NOT SIMILAR), operating on three character strings and returning a Boolean. SIMILAR determines whether or not a character string "matches" a given "pattern" (also a character string). The pattern is in the form of a "regular expression". In this regular expression, certain characters (<left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>)

have a special meaning. The optional third argument specifies the "escape character", for use when one of the special characters or the "escape character" itself is required in the pattern without its special meaning.

4.2.3.3 Operations involving large object character strings

Large object character strings cannot be operated on by all string operations. Large object character strings can, however, be operated on by the following operations:

— <null predicate>.

— <like predicate>.

— <similar predicate>.

— <position expression>.

— <comparison predicate> with an <equals operator> or <not equals operator>.

— <quantified comparison predicate> with the <equals operator> or <not equals operator>.

As a result of these restrictions, large object character strings cannot be used in (among other places):

— predicates other than those listed above and the <exists predicate>

— <general set function>.

— <group by clause>.

— <order by clause>.

— <unique constraint definition>.

— <referential constraint definition>.

— <select list> of a <query specification> that has a <set quantifier> of DISTINCT.

— UNION, INTERSECT, and EXCEPT.

— columns used for matching when forming a <joined table>.

All the operations described within Subclause 4.2.3.1, "Operators that operate on character strings and return character strings", and Subclause 4.2.3.2, "Other operators involving character strings", are supported for large object character strings.

4.2.4 Character repertoires

An SQL-implementation supports one or more character repertoires. These character repertoires may be defined by a standard or be implementation-defined.

A character repertoire is described by a character repertoire descriptor. A character repertoire descriptor includes:

— The name of the character repertoire.

— The name of the default collation for the character repertoire.

The following character repertoire names are specified as part of ISO/IEC 9075:

— SQL_CHARACTER is a character repertoire that consists of the 88 <SQL language character>s as specified in Subclause 5.1, "<SQL terminal character>". The name of the default collation is SQL_CHARACTER.

— GRAPHIC_IRV is the character repertoire that consists of the 95-character graphic subset of the International Reference Version (IRV) as specified in ISO 646:1991. Its repertoire is a proper superset of that of SQL_CHARACTER. The name of the default collation is GRAPHIC_IRV.

— LATIN1 is the character repertoire defined in ISO 8859-1. The name of the default collation is LATIN1.

— ISO8BIT is the character repertoires formed by combining the character repertoire specified by ISO 8859-1 and the "control characters" specified by ISO 6429. The repertoire consists of all 255 characters, each consisting of exactly 8 bits, as, including all control characters and all graphic characters except the character corresponding to the numeric value 0 (zero). The name of the default collation is ISO8BIT.

— UCS is the Universal Character Set repertoire specified by The Unicode Standard Version 3.1 and by ISO/IEC 10646. It is implementation-defined whether the name of the default collation is UCS_BASIC or UNICODE.

— SQL_TEXT is a character repertoire that is an implementation-defined subset of the repertoire of the Universal Character Set that includes every <SQL language character> and every character in every character set supported by the SQL-implementation. The name of the default collation is SQL_TEXT.

— SQL_IDENTIFIER is a character repertoire consisting of the <SQL language character>s and all other characters that the SQL-implementation supports for use in <regular identifier>s. The name of the default collation is SQL_IDENTIFIER.

4.2.5 Character encoding forms

An SQL-implementation supports one or more character encoding forms for each character repertoire that it supports. These character encoding forms may be defined by a standard or be implementation-defined.

A character encoding form is described by a character encoding form descriptor. A character encoding form descriptor includes:

— The name of the character encoding form.

— The name of the character repertoire to which it is applicable.

The following character encoding form names are specified as part of ISO/IEC 9075:

— SQL_CHARACTER is an implementation-defined character encoding form. It is applicable to the SQL_CHARACTER character repertoire.

— GRAPHIC_IRV is the character encoding form in which the coded representation of each character is specified in ISO 646:1991. It is applicable to the GRAPHIC_IRV character repertoire.

— LATIN1 is the character encoding form specified in ISO 8859-1. It is applicable to the LATIN1 character repertoire.

— ISO8BIT is the character encoding form specified in ISO 8859-1, augmented by ISO 6429. When restricted to the LATIN1 characters, it is the same character encoding form as LATIN1. It is applicable to the ISO8BIT character repertoire.

— UTF32 is the character encoding form specified in the Unicode Standard Annex #19, "UTF-32", in which each character is encoded as four octets. It is applicable to the UCS character repertoire.

— UTF16 is the character encoding form specified in ISO/IEC 10646-1, Annex C (normative), "Transformation format for 16 planes of Group 00 (UTF-16)", in which each character is encoded as two or four octets. It is applicable to the UCS character repertoire.

— UTF8 is the character encoding form specified in ISO/IEC 10646-1, Annex D (normative), "UCS Transformation Format 8 (UTF-8)", in which each character is encoded as from one to four octets. It is applicable to the UCS character repertoire.

— SQL_TEXT is an implementation-defined character encoding form. It is applicable to the SQL_TEXT character repertoire.

— SQL_IDENTIFIER is an implementation-defined character encoding form. It is applicable to the SQL_IDENTIFIER character repertoire.

If an SQL-implementation supplies more than one character encoding form for a particular character repertoire, then it shall specify a precedence ordering of the character encoding forms of that character repertoire. The precedence of character encoding forms applicable to the UCS character repertoire and defined in this part of ISO/IEC 9075 is:

UTF8 ≺ UTF16 ≺ UTF32

4.2.6 Collations

An SQL-implementation supports one or more collations for each character repertoire that it supports, and one or more collations for each character set that it supports. A collation is described by a collation descriptor. A collation descriptor includes:

— The name of the collation.

— The name of the character repertoire to which it is applicable.

— A list of the names of the character sets to which the collation can be applied.

— Whether the collation has the NO PAD or the PAD SPACE characteristic.

The supported collation names are specified as part of ISO/IEC 9075:

— SQL_CHARACTER is an implementation-defined collation. It is applicable to the SQL_CHARACTER character repertoire.

— GRAPHIC_IRV is a collation in which the ordering is determined by treating the code points defined by ISO 646:1991 as unsigned integers. It is applicable to the GRAPHIC_IRV character repertoire.

— LATIN1 is a collation in which the ordering is determined by treating the code points defined by ISO 8859-1 as unsigned integers. It is applicable to the LATIN1 character repertoire.

— ISO8BIT is a collation in which the ordering is determined by treating the code points defined by ISO 8859-1 as unsigned integers. When restricted to the LATIN1 characters, it produces the same collation as LATIN1. It is applicable to the ISO8BIT character repertoire.

— UCS_BASIC is a collation in which the ordering is determined entirely by the Unicode scalar values of the characters in the strings being sorted. It is applicable to the UCS character repertoire. Since every character repertoire is a subset of the UCS repertoire, the UCS_BASIC collation is potentially applicable to every character set.

NOTE 11 — The Unicode scalar value of a character is its code point treated as an unsigned integer.

— UNICODE is the collation in which the ordering is determined by applying the Unicode Collation Algorithm with the Default Unicode Collation Element Table, as specified in [Unicode10]. It is applicable to the UCS character repertoire. Since every character repertoire is a subset of the UCS repertoire, the UNICODE collation is potentially applicable to every character set.

— SQL_TEXT is an implementation-defined collation. It is applicable to the SQL_TEXT character repertoire.

— SQL_IDENTIFIER is an implementation-defined collation. It is applicable to the SQL_IDENTIFIER character repertoire.

4.2.7 Character sets

An SQL <character set specification> allows a reference to a character set name defined by a standard, an SQL-implementation, or a user.

A character set is described by a character set descriptor. A character set descriptor includes:

— The name of the character set.

— The name of the character repertoire for the character set.

— The name of the character encoding form for the character set.

— The name of the default collation for the character set.

The following SQL supported character set names are specified as part of ISO/IEC 9075:

— SQL_CHARACTER is a character set whose repertoire is SQL_CHARACTER and whose character encoding form is SQL_CHARACTER. The name of its default collation is SQL_CHARACTER.

— GRAPHIC_IRV is a character set whose repertoire is GRAPHIC_IRV and whose character encoding form is GRAPHIC_IRV. The name of its default collation is GRAPHIC_IRV.

— ASCII_GRAPHIC is a synonym for GRAPHIC_IRV.

— LATIN1 is a character set whose repertoire is LATIN1 and whose character encoding form is LATIN1. The name of its default collation is LATIN1.

— ISO8BIT is a character set whose repertoire is ISO8BIT and whose character encoding form is ISO8BIT. The name of its default collation is ISO8BIT.

— ASCII_FULL is a synonym for ISO8BIT.

— UTF32 is a character set whose repertoire is UCS and whose character encoding form is UTF32. It is implementation-defined whether the name of its default collation is UCS_BASIC or UNICODE.

— UTF16 is a character set whose repertoire is UCS and whose character encoding form is UTF16. It is implementation-defined whether the name of its default collation is UCS_BASIC or UNICODE.

— UTF8 is the name of a character set whose repertoire is UCS and whose character encoding form is UTF8. It is implementation-defined whether the name of its default collation is UCS_BASIC or UNICODE.

— SQL_TEXT is a character set whose repertoire is SQL_TEXT and whose character encoding form is SQL_TEXT. The name of its default collation is SQL_TEXT.

— SQL_IDENTIFIER is a character set whose repertoire is SQL_IDENTIFIER and whose character encoding form is SQL_IDENTIFIER. The name of its default collation is SQL_IDENTIFIER.

The result of evaluating a character string expression whose most specific type has character set *CS* is constrained to consist of characters drawn from the character repertoire of *CS*.

Table 1 — Overview of character sets

Character Set	Character Repertoire	Character Encoding Form	Collation	Synonym
GRAPHIC_IRV	GRAPHIC_IRV	GRAPHIC_IRV	GRAPHIC_IRV	ASCII_GRAPHIC
ISO8BIT	ISO8BIT	ISO8BIT	ISO8BIT	ASCII_FULL
LATIN1	LATIN1	LATIN1	LATIN1	
SQL_CHARAC-TER	SQL_CHARAC-TER	SQL_CHARAC-TER	SQL_CHARAC-TER	
SQL_TEXT	SQL_TEXT	SQL_TEXT	SQL_TEXT	
SQL-IDENTI-FIER	SQL-IDENTI-FIER	SQL-IDENTI-FIER	SQL-IDENTI-FIER	
UTF16	UCS	UTF16	UCS_BASIC or UNICODE	
UTF32	UCS	UTF32	UCS_BASIC or UNICODE	
UTF8	UCS	UTF8	UCS_BASIC or UNICODE	

NOTE 12 — An SQL-implementation may supply additional character sets and/or additional character encoding forms and collations for character sets defined in this Part of ISO/IEC 9075.

4.2.8 Universal character sets

A *UCS string* is a character string whose character repertoire is UCS and whose character encoding form is one of UTF8, UTF16, or UTF32. Any two UCS strings are comparable.

An SQL-implementation may assume that all UCS strings are normalized in Normalization Form C (NFC), as specified by [Unicode15]. With the exception of <normalize function> and <normalized predicate>, the result of any operation on an unnormalized UCS string is implementation-defined.

Conversion of UCS strings from one character set to another is automatic.

Detection of a noncharacter in a UCS-string causes an exception condition to be raised. The detection of an unassigned code point does not.

4.3 Binary strings

4.3.1 Introduction to binary strings

A binary string is a sequence of octets that does not have either a character set or collation associated with it.

A binary string data type is described by a binary string data type descriptor. A binary string data type descriptor contains:

— The name of the data type (BINARY LARGE OBJECT).

— The maximum length of the binary string data type (in octets).

A binary string is assignable only to sites of data type BINARY LARGE OBJECT. If a store assignment would result in the loss of non-zero octets due to truncation, then an exception condition is raised. If a retrieval assignment would result in the loss of octets due to truncation, then a warning condition is raised.

4.3.2 Binary string comparison

All binary string values are comparable. When binary string values are compared, they shall have exactly the same length (in octets) to be considered equal. Binary string values can be compared only for equality.

4.3.3 Operations involving binary strings

4.3.3.1 Operators that operate on binary strings and return binary strings

<blob concatenation> is an operator, | |, that returns a binary string by joining its binary string operands in the order given.

<blob substring function> is a triadic function identical in syntax and semantics to <character substring function> except that the returned value is a binary string.

<blob overlay function> is a function identical in syntax and semantics to <character overlay function> except that the first argument, second argument, and returned value are all binary strings.

<trim function> when applied to binary strings is identical in syntax (apart from the default <trim character>) and semantics to the corresponding operation on character strings except that the returned value is a binary string.

4.3.3.2 Other operators involving binary strings

<length expression> returns the length of a given binary string, as an exact numeric value, in characters or octets according to the choice of function.

<position expression> when applied to binary strings is identical in syntax and semantics to the corresponding operation on character strings except that the operands are binary strings.

<like predicate> when applied to binary strings is identical in syntax and semantics to the corresponding operation on character strings except that the operands are binary strings.

Binary strings cannot be used in:

— Predicates other than <comparison predicate> with an <equals operator> or a <not equals operator>, <quantified comparison predicate> with an <equals operator> or a <not equals operator>, and <exists predicate>.

— <general set function>.

— <group by clause>.

— <order by clause>.

— <unique constraint definition>.

— <select list> of a <query specification> that has a <set quantifier> of DISTINCT.

— UNION, INTERSECT, and EXCEPT.

— Columns used for matching when forming a <joined table>.

4.4 Numbers

4.4.1 Introduction to numbers

A number is either an exact numeric value or an approximate numeric value. Any two numbers are comparable.

A numeric type is described by a numeric type descriptor. A numeric type descriptor contains:

— The name of the specific numeric type (NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, or DOUBLE PRECISION).

— The precision of the numeric type.

— The scale of the numeric type, if it is an exact numeric type.

— An indication of whether the precision (and scale) are expressed in decimal or binary terms.

An SQL-implementation is permitted to regard certain <exact numeric type>s as equivalent, if they have the same precision, scale, and radix, as permitted by the Syntax Rules of Subclause 6.1, "<data type>". When two or more <exact numeric type>s are equivalent, the SQL-implementation chooses one of these equivalent <exact numeric type>s as the *normal form* representing that equivalence class of <exact numeric type>s. The normal form determines the name of the exact numeric type in the numeric type descriptor.

Similarly, an SQL-implementation is permitted to regard certain <approximate numeric type>s as equivalent, as permitted the Syntax Rules of Subclause 6.1, "<data type>", in which case the SQL-implementation chooses a *normal form* to represent each equivalence class of <approximate numeric type> and the normal form determines the name of the approximate numeric type.

For every numeric type, the least value is less than zero and the greatest value is greater than zero.

4.4.2 Characteristics of numbers

An exact numeric type has a precision P and a scale S. P is a positive integer that determines the number of significant digits in a particular radix R, where R is either 2 or 10. S is a non-negative integer. Every value of an exact numeric type of scale S is of the form $n \times 10^{-S}$, where n is an integer such that $-R^P \leq n < R^P$.

NOTE 13 — Not every value in that range is necessarily a value of the type in question.

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 that specifies the magnitude of the mantissa. An approximate numeric value has a precision. The precision is a positive integer that specifies the number of significant binary digits in the mantissa. The value of an approximate numeric value is the mantissa multiplied by a factor determined by the exponent.

An <approximate numeric literal> *ANL* consists of an <exact numeric literal> (called the <mantissa>), the letter 'E' or 'e', and a <signed integer> (called the <exponent>). If M is the value of the <mantissa> and E is the value of the <exponent>, then $M * 10^E$ is the *apparent value* of *ANL*. The actual value of *ANL* is approximately the apparent value of *ANL*, according to implementation-defined rules.

A number is assignable only to sites of numeric type. If an assignment of some number would result in a loss of its most significant digit, an exception condition is raised. If least significant digits are lost, implementation-defined rounding or truncating occurs, with no exception condition being raised. The rules for arithmetic are specified in Subclause 6.26, "<numeric value expression>".

Whenever an exact or approximate numeric value is assigned to an exact numeric value site, an approximation of its value that preserves leading significant digits after rounding or truncating is represented in the declared type of the target. The value is converted to have the precision and scale of the target. The choice of whether to truncate or round is implementation-defined.

An approximation obtained by truncation of a numeric value *N* for an <exact numeric type> *T* is a value *V* in *T* such that *N* is not closer to zero than is *V* and there is no value in *T* between *V* and *N*.

An approximation obtained by rounding of a numeric value *N* for an <exact numeric type> *T* is a value *V* in *T* such that the absolute value of the difference between *N* and the numeric value of *V* is not greater than half the absolute value of the difference between two successive numeric values in *T*. If there is more than one such value *V*, then it is implementation-defined which one is taken.

All numeric values between the smallest and the largest value, inclusive, in a given exact numeric type have an approximation obtained by rounding or truncation for that type; it is implementation-defined which other numeric values have such approximations.

An approximation obtained by truncation or rounding of a numeric value *N* for an <approximate numeric type> *T* is a value *V* in *T* such that there is no numeric value in *T* and distinct from that of *V* that lies between the numeric value of *V* and *N*, inclusive.

If there is more than one such value *V* then it is implementation-defined which one is taken. It is implementation-defined which numeric values have approximations obtained by rounding or truncation for a given approximate numeric type.

Whenever an exact or approximate numeric value is assigned to an approximate numeric value site, an approximation of its value is represented in the declared type of the target. The value is converted to have the precision of the target.

Operations on numbers are performed according to the normal rules of arithmetic, within implementation-defined limits, except as provided for in Subclause 6.26, "<numeric value expression>".

4.4.3 Operations involving numbers

As well as the usual arithmetic operators, plus, minus, times, divide, unary plus, and unary minus, there are the following functions that return numbers:

— <position expression> (see Subclause 4.2.3, "Operations involving character strings", and Subclause 4.3.3, "Operations involving binary strings") takes two strings as arguments and returns an integer.

— <length expression> (see Subclause 4.2.3, "Operations involving character strings", and Subclause 4.3.3, "Operations involving binary strings") operates on a string argument and returns an integer.

— <extract expression> (see Subclause 4.6.4, "Operations involving datetimes and intervals") operates on a datetime or interval argument and returns an exact numeric.

— <cardinality expression> (see Subclause 4.10.5, "Operations involving arrays", and Subclause 4.10.6, "Operations involving multisets") operates on a collection argument and returns an integer.

— <absolute value expression> operates on a numeric argument and returns its absolute value in the same most specific type.

— <modulus expression> operates on two exact numeric arguments with scale 0 (zero) and returns the modulus (remainder) of the first argument divided by the second argument as an exact numeric with scale 0 (zero).

— <natural logarithm> computes the natural logarithm of its argument.

— <exponential function> computes the exponential function, that is, *e*, (the base of natural logarithms) raised to the power equal to its argument.

— <power function> raises its first argument to the power of its second argument.

— <square root> computes the square root of its argument.

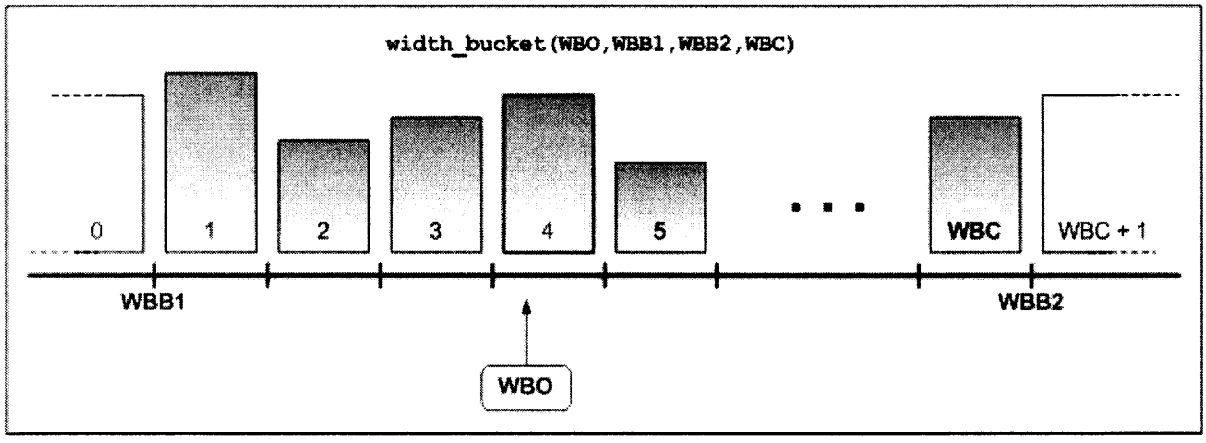
— <floor function> computes the greatest integer less than or equal to its argument.

— <ceiling function> computes the least integer greater than or equal to its argument.

— <width bucket function> is a function of four arguments, returning an integer between 0 (zero) and the value of the final argument plus 1 (one), by assigning the first argument to an equi-width partitioning of the range of numbers between the second and third arguments. Values outside the range between the second and third arguments are assigned to either 0 (zero) or the value of the final argument plus 1 (one).

NOTE 14 — The semantics of <width bucket function> are illustrated in Figure 2, "Illustration of WIDTH_BUCKET Semantics".

Figure 2 — Illustration of WIDTH_BUCKET Semantics



4.5 Boolean types

4.5.1 Introduction to Boolean types

The data type boolean comprises the distinct truth values *True* and *False*. Unless prohibited by a NOT NULL constraint, the boolean data type also supports the truth value *Unknown* as the null value. This specification does not make a distinction between the null value of the boolean data type and the truth value *Unknown* that is the result of an SQL <predicate>, <search condition>, or <boolean value expression>; they may be used interchangeably to mean exactly the same thing.

The boolean data type is described by the boolean data type descriptor. The boolean data type descriptor contains:

— The name of the boolean data type (BOOLEAN).

4.5.2 Comparison and assignment of booleans

All boolean values and SQL truth values are comparable and all are assignable to a site of type boolean. The value *True* is greater than the value *False*, and any comparison involving the null value or an *Unknown* truth value will return an *Unknown* result. The values *True* and *False* may be assigned to any site having a boolean data type; assignment of *Unknown*, or the null value, is subject to the nullability characteristic of the target.

4.5.3 Operations involving booleans

4.5.3.1 Operations on booleans that return booleans

The monadic boolean operator NOT and the dyadic boolean operators AND and OR take boolean operands and produce a boolean result (see Table 11, "Truth table for the AND boolean operator", and Table 12, "Truth table for the OR boolean operator").

4.5.3.2 Other operators involving booleans

Every SQL <predicate>, <search condition>, and <boolean value expression> may be considered as an operator that returns a boolean result.

4.6 Datetimes and intervals

4.6.1 Introduction to datetimes and intervals

A datetime data type is described by a datetime data type descriptor. An interval data type is described by an interval data type descriptor.

A datetime data type descriptor contains:

— The name of the specific datetime data type (DATE, TIME WITHOUT TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE).

— The value of the <time fractional seconds precision>, if it is a TIME WITHOUT TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE type.

An interval data type descriptor contains:

— The name of the interval data type (INTERVAL).

— An indication of whether the interval data type is a year-month interval or a day-time interval.

— The <interval qualifier> that describes the precision of the interval data type.

A value described by an interval data type descriptor is always signed.

Every datetime or interval data type has an implied *length in positions*. Let *D* denote a value in some datetime or interval data type *DT*. The length in positions of *DT* is constant for all *D*. The length in positions is the number of characters from the character set SQL_TEXT that it would take to represent any value in a given datetime or interval data type.

An approximation obtained by rounding of a datetime or interval value *D* for a <datetime type> or <interval type> *T* is a value *V* in *T* such that the absolute value of the difference between *D* and the numeric value of *V* is not greater than half the absolute value of the difference between two successive datetime or interval values in *T*. If there is more than one such value *V*, then it is implementation-defined which one is taken.

4.6.2 Datetimes

Table 2, "Fields in datetime values", specifies the fields that can make up a datetime value; a datetime value is made up of a subset of those fields. Not all of the fields shown are required to be in the subset, but every field that appears in the table between the first included primary field and the last included primary field shall also be included. If either time zone field is in the subset, then both of them shall be included.

Table 2 — Fields in datetime values

Keyword	Meaning
YEAR	Year
MONTH	Month within year
DAY	Day within month
HOUR	Hour within day
MINUTE	Minute within hour
SECOND	Second and possibly fraction of a second within minute
TIMEZONE_HOUR	Hour value of time zone displacement
TIMEZONE_MINUTE	Minute value of time zone displacement

There is an ordering of the significance of <primary datetime field>s. This is, from most significant to least significant: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

The <primary datetime field>s other than SECOND contain non-negative integer values, constrained by the natural rules for dates using the Gregorian calendar. SECOND, however, can be defined to have a <time fractional seconds precision> that indicates the number of decimal digits maintained following the decimal point in the seconds value, a non-negative exact numeric value.

There are three classes of datetime data types defined within this part of ISO/IEC 9075:

— DATE — contains the <primary datetime field>s YEAR, MONTH, and DAY.

— TIME — contains the <primary datetime field>s HOUR, MINUTE, and SECOND.

— TIMESTAMP — contains the <primary datetime field>s YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

Items of type datetime are comparable only if they have the same <primary datetime field>s.

A datetime data type that specifies WITH TIME ZONE is a data type that is *datetime with time zone*, while a datetime data type that specifies WITHOUT TIME ZONE is a data type that is *datetime without time zone*.

The surface of the earth is divided into zones, called time zones, in which every correct clock tells the same time, known as *local time*. Local time is equal to UTC (Coordinated Universal Time) plus the *time zone displacement*, which is an interval value that ranges between INTERVAL '–12:59' HOUR TO MINUTE and INTERVAL '+14:00' HOUR TO MINUTE. The time zone displacement is constant throughout a time zone, changing at the beginning and end of Daylight Time, where applicable.

A datetime value, of data type TIME WITHOUT TIME ZONE or TIMESTAMP WITHOUT TIME ZONE, may represent a local time, whereas a datetime value of data type TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE represents UTC.

On occasion, UTC is adjusted by the omission of a second or the insertion of a "leap second" in order to maintain synchronization with sidereal time. This implies that sometimes, but very rarely, a particular minute will contain exactly 59, 61, or 62 seconds. Whether an SQL-implementation supports leap seconds, and the consequences of such support for date and interval arithmetic, is implementation-defined.

For the convenience of users, whenever a datetime value with time zone is to be implicitly derived from one without (for example, in a simple assignment operation), SQL assumes the value without time zone to be local, subtracts the current default time zone displacement of the SQL-session from it to give UTC, and associates that time zone displacement with the result.

Conversely, whenever a datetime value without time zone is to be implicitly derived from one with, SQL assumes the value with time zone to be UTC, adds the time zone displacement to it to give local time, and the result, without any time zone displacement, is local.

The preceding principles, as implemented by <cast specification>, result in data type conversions between the various datetime data types, as summarized in Table 3, "Datetime data type conversions".

Table 3 — Datetime data type conversions

	to DATE	**to TIME WITHOUT TIME ZONE**	**to TIME WITH TIME ZONE**	**to TIMESTAMP WITHOUT TIME ZONE**	**to TIMESTAMP WITH TIME ZONE**
from DATE	*trivial*	*not supported*	*not supported*	Copy year, month, and day; set hour, minute, and second to 0 (zero)	*SV* ⇒ TSw/oTZ ⇒ TSw/TZ
from TIME WITHOUT TIME ZONE	*not supported*	*trivial*	*TV.UTC* = *SV* – *STZD* (modulo 24); *TV.TZ* = *STZD*	Copy date fields from CUR-RENT_DATE and time fields from *SV*	*SV* ⇒ TSw/oTZ ⇒ TSw/TZ
from TIME WITH TIME ZONE	*not supported*	*SV.UTC* + *SV.TZ* (modulo 24)	*trivial*	*SV* ⇒ TSw/TZ ⇒ TSwo/TZ	Copy date fields from CUR-RENT_DATE and time and time zone fields from *SV*

	to DATE	to TIME WITHOUT TIME ZONE	to TIME WITH TIME ZONE	to TIMESTAMP WITHOUT TIME ZONE	to TIMESTAMP WITH TIME ZONE
from TIMES-TAMP WITHOUT TIME ZONE	Copy date fields from *SV*	Copy time fields from *SV*	*SV* ⇒ TSw/TZ ⇒ TIMEw/TZ	*trivial*	*TV.UTC* = *SV* – *STZD*; *TV.TZ* = *STZD*
from TIMES-TAMP WITH TIME ZONE	*SV* ⇒ TSw/oTZ ⇒ DATE	*SV* ⇒ TSw/oTZ ⇒ TIMEw/oTZ	Copy time and time zone fields from *SV*	*SV.UTC* + *SV.TZ*	*trivial*

† **Where** *SV* is the source value, *TV* is the target value, *.UTC* is the UTC component of *SV* or *TV* (if and only if the source or target has time zone), *STZD* is the SQL-session default time zone displacement, ⇒ means to cast from the type preceding the arrow to the type following the arrow, "TIMEw/TZ" is "TIME WITH TIME ZONE", "TIMEw/oTZ" is "TIME WITHOUT TIME ZONE", "TSw/TZ" is "TIMESTAMP WITH TIME ZONE", and "TSw/oTZ" is "TIMESTAMP WITHOUT TIME ZONE".

A datetime is assignable to a site only if the source and target of the assignment are both of type DATE, or both of type TIME (regardless whether WITH TIME ZONE or WITHOUT TIME ZONE is specified or implicit), or both of type TIMESTAMP (regardless whether WITH TIME ZONE or WITHOUT TIME ZONE is specified or implicit).

4.6.3 Intervals

There are two classes of intervals. One class, called *year-month* intervals, has an express or implied datetime precision that includes no fields other than YEAR and MONTH, though not both are required. The other class, called *day-time* intervals, has an express or implied interval precision that can include any fields other than YEAR or MONTH.

Table 4, "Fields in year-month INTERVAL values", specifies the fields that make up a year-month interval. A year-month interval is made up of a contiguous subset of those fields.

Table 4 — Fields in year-month INTERVAL values

Keyword	Meaning
YEAR	Years
MONTH	Months

Table 5, "Fields in day-time INTERVAL values", specifies the fields that make up a day-time interval. A day-time interval is made up of a contiguous subset of those fields.

Table 5 — Fields in day-time INTERVAL values

Keyword	Meaning
DAY	Days
HOUR	Hours
MINUTE	Minutes
SECOND	Seconds and possibly fractions of a second

The actual subset of fields that comprise a value of either type of interval is defined by an <interval qualifier> and this subset is known as the precision of the value.

Within a value of type interval, the first field is constrained only by the <interval leading field precision> of the associated <interval qualifier>. Table 6, "Valid values for fields in INTERVAL values", specifies the constraints on subsequent field values.

Table 6 — Valid values for fields in INTERVAL values

Keyword	Valid values of INTERVAL fields
YEAR	Unconstrained except by <interval leading field precision>
MONTH	Months (within years) (0-11)
DAY	Unconstrained except by <interval leading field precision>
HOUR	Hours (within days) (0-23)
MINUTE	Minutes (within hours) (0-59)
SECOND	Seconds (within minutes) (0-59.999...)

Values in interval fields other than SECOND are integers and have precision 2 when not the first field. SECOND, however, can be defined to have an <interval fractional seconds precision> that indicates the number of decimal digits maintained following the decimal point in the seconds value. When not the first field, SECOND has a precision of 2 places before the decimal point.

Fields comprising an item of type interval are also constrained by the definition of the Gregorian calendar.

Year-month intervals are comparable only with other year-month intervals. If two year-month intervals have different interval precisions, they are, for the purpose of any operations between them, effectively converted to the same precision by appending new <primary datetime field>s to either the most significant end of one

interval, the least significant end of one interval, or both. New least significant <primary datetime field>s are assigned a value of 0 (zero). When it is necessary to add new most significant datetime fields, the associated value is effectively converted to the new precision in a manner obeying the natural rules for dates and times associated with the Gregorian calendar.

Day-time intervals are comparable only with other day-time intervals. If two day-time intervals have different interval precisions, they are, for the purpose of any operations between them, effectively converted to the same precision by appending new <primary datetime field>s to either the most significant end of one interval or the least significant end of one interval, or both. New least significant <primary datetime field>s are assigned a value of 0 (zero). When it is necessary to add new most significant datetime fields, the associated value is effectively converted to the new precision in a manner obeying the natural rules for dates and times associated with the Gregorian calendar.

4.6.4 Operations involving datetimes and intervals

Table 7, "Valid operators involving datetimes and intervals", specifies the declared types of arithmetic expressions involving datetime and interval operands.

Table 7 — Valid operators involving datetimes and intervals

Operand 1	Operator	Operand 2	Result Type
Datetime	–	Datetime	Interval
Datetime	+ or –	Interval	Datetime
Interval	+	Datetime	Datetime
Interval	+ or –	Interval	Interval
Interval	* or /	Numeric	Interval
Numeric	*	Interval	Interval

Arithmetic operations involving values of type datetime or interval obey the natural rules associated with dates and times and yield valid datetime or interval results according to the Gregorian calendar.

Operations involving values of type datetime require that the datetime values be comparable. Operations involving values of type interval require that the interval values be comparable.

Operations involving a datetime and an interval preserve the time zone of the datetime operand. If the datetime operand does not include a time zone displacement, then the result has no time zone displacement.

<overlaps predicate> uses the operator OVERLAPS to determine whether or not two chronological periods overlap in time. A chronological period is specified either as a pair of datetimes (starting and ending) or as a starting datetime and an interval. If the length of the period is greater than 0 (zero), then the period consists of all points of time greater than or equal to the lower endpoint, and less than the upper endpoint. If the length of

the period is equal to 0 (zero), then the period consists of a single point in time, the lower endpoint. Two periods overlap if they have at least one point in common.

<extract expression> operates on a datetime or interval and returns an exact numeric value representing the value of one component of the datetime or interval.

<interval absolute value function> operates on an interval argument and returns its absolute value in the same most specific type.

4.7 User-defined types

4.7.1 Introduction to user-defined types

A user-defined type is a schema object, identified by a <user-defined type name>. The definition of a user-defined type specifies a representation for values of that type. In some cases, known as *distinct types*, the representation is a single predefined type, known as the *source type*; in others, *structured types*, it consists of a list of attribute definitions. Although the attribute definitions are said to define the representation of the user-defined type, in fact they implicitly define certain functions (observers and mutators) that are part of the interface of the user-defined type; physical representations of user-defined type values are implementation-dependent.

The definition of a user-defined type may include a <method specification list> consisting of one or more <method specification>s. A <method specification> is either an <original method specification> or an <overriding method specification> (in which case the type being defined must be a structured type). Each <original method specification> specifies:

— The <method name>.

— The <specific method name>.

— The <SQL parameter declaration list>.

— The <returns data type>.

— The <result cast from type> (if any).

— Whether the method is type-preserving.

— The <language clause>.

— If the language is not SQL, then the <parameter style>.

— Whether STATIC or CONSTRUCTOR is specified.

— Whether the method is deterministic.

— Whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

— Whether the method should be evaluated as the null value whenever any argument is the null value, without actually invoking the method.

Each <overriding method specification> specifies the <method name>, the <specific method name>, the <SQL parameter declaration list> and the <returns data type>. For each <overriding method specification>, there shall be an <original method specification> with the same <method name> and <SQL parameter declaration list> in some proper supertype of the user-defined type. Every SQL-invoked method in a schema shall correspond to exactly one <original method specification> or <overriding method specification> associated with some user-defined type existing in that schema.

A method *M* that corresponds to an <original method specification> in the definition of a structured type *T1* is an *original method* of *T1*. A method *M* that corresponds to an <overriding method specification> in the definition of *T1* is an *overriding method* of *T1*.

A method *M* is a *method of type T1* if one of the following holds:

— *M* is an original method of *T1*.

— *M* is an overriding method of *T1*.

— There is a proper supertype *T2* of *T1* such that *M* is an original or overriding method of *T2* and such that there is no method *M3* such that *M3* has the same <method name> and <SQL parameter declaration list> as *M* and *M3* is an original method or overriding method of a type *T3* such that *T2* is a proper supertype of *T3* and *T3* is a supertype of *T1*.

If *T1* is a subtype of *T2* and *M1* is a method of *T1* such that there exists a method *M2* of *T2* such that *M1* and *M2* have the same <method name> and the same unaugmented <SQL parameter declaration list>, then *M1* is an *inherited method* of *T1* from *T2*.

4.7.2 User-defined type descriptor

A user-defined type is described by a user-defined type descriptor. A user-defined type descriptor contains:

— The name of the user-defined type (<user-defined type name>). This is the type designator of that type, used in type precedence lists (see Subclause 9.5, "Type precedence list determination").

— An indication of whether the user-defined type is a structured type or a distinct type.

— The ordering form for the user-defined type (EQUALS, FULL, or NONE).

— The ordering category for the user-defined type (RELATIVE, MAP, or STATE).

— A <specific routine designator> identifying the ordering function, depending on the ordering category.

— If the user-defined type is a direct subtype of another user-defined type, then the name of that user-defined type.

— If the representation is a predefined data type, then the descriptor of that type; otherwise the attribute descriptor of every originally-defined attribute and every inherited attribute of the user-defined type.

— An indication of whether the user-defined type is instantiable or not instantiable.

— An indication of whether the user-defined type is final or not final.

— The transform descriptor of the user-defined type.

— If the user-defined type is a structured type, then:

- Whether the referencing type of the structured type has a user-defined representation, a derived representation, or a system-defined representation.
- If user-defined representation is specified, then the type descriptor of the representation type of the referencing type of the structured type; otherwise, if derived representation is specified, then the list of attributes.

NOTE 15 — "user-defined representation", "derived representation", and "system-defined representation" of a reference type are defined in Subclause 4.9, "Reference types".

— If the <method specification list> is specified, then for each <method specification> contained in <method specification list>, a *method specification descriptor* that includes:

- The <method name>.
- The <specific method name>.
- The <SQL parameter declaration list> augmented to include the implicit first parameter with parameter name SELF.
- The <language name>.
- If the <language name> is not SQL, then the <parameter style>.
- The <returns data type>.
- The <result cast from type>, if any.
- An indication as to whether the <method specification> is an <original method specification> or an <overriding method specification>.
- If the <method specification> is an <original method specification>, then an indication of whether STATIC or CONSTRUCTOR is specified.
- An indication whether the method is deterministic.
- An indication whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.
- An indication whether the method should not be invoked if any argument is the null value, in which case the value of the method is the null value.

NOTE 16 — The characteristics of an <overriding method specification> other than the <method name>, <SQL parameter declaration list>, and <returns data type> are the same as the characteristics for the corresponding <original method specification>.

4.7.3 Observers and mutators

Corresponding to every attribute of every structured type is exactly one implicitly-defined observer function and exactly one implicitly-defined mutator function. These are both SQL-invoked functions. Further, the mutator function is a type-preserving function.

Let *A* be the name of an attribute of structured type *T* and let *AT* be the data type of *A*. The signature of the observer function for this attribute is FUNCTION *A*(*T*) and its result data type is *AT*. The signature of the mutator function for this attribute is FUNCTION *A*(*T* RESULT, *AT*) and its result data type is *T*.

Let *V* be a value in data type *T* and let *AV* be a value in data type *AT*. The invocation *A*(*V*,*AV*) returns *MV* such that "*A*(*MV*) is identical to *AV*" and for every attribute *A'* (*A'* ≠ *A*) of *T*, "*A'*(*MV*) is identical to *A'*(*V*)". The most specific type of *MV* is the most specific type of *V*.

4.7.4 Constructors

Associated with each structured type *ST* is one implicitly defined *constructor function*, if and only if *ST* is instantiable.

Let *TN* be the name of a structured type *T*. The signature of the constructor function for *T* is *TN*() and its result data type is *T*. The invocation *TN*() returns a value *V* such that *V* is not null and, for every attribute *A* of *T*, *A*(*V*) returns the default value of *A*. The most specific type of *V* is *T*.

For every structured type *ST* that is instantiable, zero or more SQL-invoked constructor methods can be specified. The names of those methods shall be equivalent to the name of the type for which they are specified.

NOTE 17 — SQL-invoked constructor methods are original methods that cannot be overloaded. An SQL-invoked constructor method and a regular SQL-invoked function may exist such that they have equivalent routine names, the types of the first parameter of the method's augmented parameter list and the function's parameter list are the same, and the types of the corresponding remaining parameters (if any) are identical according to the Syntax Rules of Subclause 9.16, "Data type identity".

4.7.5 Subtypes and supertypes

As a consequence of the <subtype clause> of <user-defined type definition>, two structured types T_a and T_b that are not compatible can be such that T_a is a subtype of T_b. See Subclause 11.41, "<user-defined type definition>".

A type T_a is a *direct subtype* of a type T_b if T_a is a proper subtype of T_b and there does not exist a type T_c such that T_c is a proper subtype of T_b and a proper supertype of T_a.

A type T_a is a subtype of type T_b if one of the following pertains:

— T_a and T_b are compatible;

— T_a is a direct subtype of T_b; or

— T_a is a subtype of some type T_c and T_c is a direct subtype of T_b.

By the same token, T_b is a supertype of T_a and is a direct supertype of T_a in the particular case where T_a is a direct subtype of T_b.

If T_a is a subtype of T_b and T_a and $T_b$are not compatible, then T_a is proper subtype of T_b and T_b is a proper supertype of T_a. A type cannot be a proper supertype of itself.

A type with no proper supertypes is a maximal supertype. A type with no proper subtypes is a leaf type.

Let T_a be a maximal supertype and let T be a subtype of T_a. The set of all subtypes of T_a (which includes T_a itself) is called a *subtype family* of T or (equivalently) of T_a. A subtype family is not permitted to have more than one maximal supertype.

Every value in a type T is a value in every supertype of T. A value V in type T has exactly one most specific type MST such that MST is a subtype of T and V is not a value in any proper subtype of MST. The most specific type of value need not be a leaf type. For example, a type structure might consist of a type PERSON that has STUDENT and EMPLOYEE as its two subtypes, while STUDENT has two direct subtypes UG_STUDENT and PG_STUDENT. The invocation STUDENT() of the constructor function for STUDENT returns a value whose most specific type is STUDENT, which is not a leaf type.

If T_a is a subtype of T_b, then a value in T_a can be used wherever a value in T_b is expected. In particular, a value in T_a can be stored in a column of type T_b, can be substituted as an argument for an input SQL parameter of data type T_b, and can be the value of an invocation of an SQL-invoked function whose result data type is T_b.

A type T is said to be the *minimal common supertype* of a set of types S if T is a supertype of every type in S and a subtype of every type that is a supertype of every type in S.

NOTE 18 — Because a subtype family has exactly one maximal supertype, if two types have a common subtype, they shall also have a minimal common supertype. Thus, for every set of types drawn from the same subtype family, there is some member of that family that is the minimal common supertype of all of the types in that set.

If a structured type ST is defined to be not instantiable, then the most specific type of every value in ST is necessarily of some proper subtype of ST.

If a user-defined type UDT is defined to be final, then UDT has no proper subtypes. As a consequence, the most specific type of every value in UDT is necessarily UDT.

Users shall have the UNDER privilege on a type before they can define any direct subtypes of it. A type can have more than one direct subtype. A user-defined type or a reference type can have at most one direct supertype. A row type can have more than one direct supertype.

A <user-defined type definition> for type T can include references to components of every direct supertype of T. Effectively, components of all direct supertype representations are copied to the subtype's representation.

4.7.6 User-defined type comparison and assignment

Let *comparison type* of a user-defined type T_a be the user-defined type T_b that satisfies all the following conditions:

1) The type designator of T_b is in the type precedence list of T_a.

2) The user-defined type descriptor of T_b includes an ordering form that is EQUALS or FULL.

3) The descriptor of no type T_c whose type designator precedes that of T_b in the type precedence list of T_a includes an ordering form that includes EQUALS or FULL.

If there is no such type T_b, then T_a has no comparison type.

Let *comparison form* of a user-defined type T_a be the ordering form included in the user-defined type descriptor of the comparison type of T_a.

Let *comparison category* of a user-defined type T_a be the ordering category included in the user-defined type descriptor of the comparison type of T_a.

Let *comparison function* of a user-defined type T_a be the ordering function included in the user-defined type descriptor of the comparison type of T_a.

Two values *V1* and *V2* whose most specific types are user-defined types *T1* and *T2* are comparable if and only if *T1* and *T2* are in the same subtype family and each have some comparison type *CT1* and *CT2*, respectively. *CT1* and *CT2* constrain the comparison forms and comparison categories of *T1* and *T2* to be the same and to be the same as those of all their supertypes. If the comparison category is either STATE or RELATIVE, then *T1* and *T2* are constrained to have the same comparison function; if the comparison category is MAP, they are not constrained to have the same comparison function.

NOTE 19 — Explicit cast functions or attribute comparisons can be used to make both values of the same subtype family or to perform the comparisons on attributes of the user-defined types.

NOTE 20 — "Subtype" and "subtype family" are defined in Subclause 4.7.5, "Subtypes and supertypes".

If there is no appropriate user-defined cast function, then an expression *E* whose declared type is some user-defined type *UDT1* is assignable to a site *S* whose declared type is some user-defined type *UDT2* if and only if *UDT1* is a subtype of *UDT2*. The effect of the assignment of *E* to *S* is that the value of *S* is *V*, obtained by the evaluation of *E*. The most specific type of *V* is some subtype of *UDT1*, possibly *UDT1* itself, while the declared type of *S* remains *UDT2*.

An expression whose declared type is some distinct type whose source type is *SDT* is assignable to any site whose declared type is *SDT* becaues of the implicit cast functions created by the General Rules of Subclause 11.41, "<user-defined type definition>". Similarly, an expression whose declared type is some pre-defined data type *SDT* is assignable to any site whose declared type is some distinct type whose source type is *SDT*.

4.7.7 Transforms for user-defined types

Transforms are SQL-invoked functions that are automatically invoked when values of user-defined types are transferred from SQL-environment to host languages or vice-versa.

A transform is associated with a user-defined type. A transform identifies a list of *transform groups* of up to two SQL-invoked functions, called the *transform functions*, each identified by a group name. The group name of a transform group is an <identifier> such that no two transform groups for a transform have the same group name. The two transform functions are:

— **from-sql function** — This SQL-invoked function maps the user-defined type value into a value of an SQL pre-defined type, and gets invoked whenever a user-defined type value is passed to a host language program or an external routine.

— **to-sql function** — This SQL-invoked function maps a value of an SQL predefined type to a value of a user-defined type and gets invoked whenever a user-defined type value is supplied by a host language program or an external routine.

A transform is defined by a <transform definition>. A transform is described by a *transform descriptor*. A transform descriptor includes a possibly empty list of *transform group descriptors*, where each transform group descriptor includes:

— The group name of the transform group.

— The specific name of the from-sql function, if any, associated with the transform group.

— The specific name of the to-sql function, if any, associated with the transform group.

4.8 Row types

A row type is a sequence of (<field name> <data type>) pairs, called *fields*. It is described by a row type descriptor. A row type descriptor consists of the field descriptor of every field of the row type.

The most specific type of a row of a table is a row type. In this case, each column of the row corresponds to the field of the row type that has the same ordinal position as the column.

Row type *RT2* is a subtype of data type *RT1* if and only if *RT1* and *RT2* are row types of the same degree and, in every n-th pair of corresponding field definitions, $FD1_n$ in *RT1* and $FD2_n$ in *RT2*, the <field name>s are equivalent and the <data type> of $FD2_n$ is a subtype of the <data type> of $FD1_n$.

A value of row type *RT1* is assignable to a site of row type *RT2* if and only if the degree of *RT1* is the same as the degree of *RT2* and every field in *RT1* is assignable to the field in the same ordinal position in *RT2*.

A value of row type *RT1* is comparable with a value of row type *RT2* if and only if the degree of *RT1* is the same as the degree of *RT2* and every field in *RT1* is comparable with the field in the same ordinal position in *RT2*.

4.9 Reference types

4.9.1 Introduction to reference types

A *REF value* is a value that references a row in a *referenceable table* (see Subclause 4.14.5, "Referenceable tables, subtables, and supertables"). A referenceable table is necessarily also a *typed table* (that is, it has an associated structured type from which its row type is derived).

Given a structured type *T*, the REF values that can reference rows in typed tables defined on *T* collectively form a certain data type *RT* known as a *reference type*. *RT* is the *referencing type* of *T* and *T* is the *referenced type* of *RT*.

Let *TN* be name of *T*. The type designator of *RT* is REF(*TN*).

Values of two reference types are comparable if the referenced types of their declared types have some common supertype.

An expression *E* whose declared type is some reference type *RT1* is assignable to a site *S* whose declared type is some reference type *RT2* if and only if the referenced type of *RT1* is a subtype of the referenced type of *RT2*. The effect of the assignment of *E* to *S* is that the value of *S* is *V*, obtained by the evaluation of *E*. The most specific type of *V* is some subtype of *RT1*, possibly *RT1* itself, while the declared type of *S* remains *RT2*.

A site *RS* that is occupied by a REF value might have a *scope*, which determines the effect of an invocation of <reference resolution> *RR* on the value at *RS*. A scope is specified as a table name *STN* and consists at any time of every row in the table *ST* identified by *STN*. *ST* is the *scoped table* of *RR*. The scope of *RS* is specified in the declared type of *RS*. If no scope is specified in the declard type of *RS*, then <reference resolution> is not available.

A reference type is described by a reference type descriptor. The reference type descriptor for *RT* includes:

— The type designator of *RT*.

— The name of the referenceable table, if any, that is the scope of *RT*.

In a host variable, a REF value is materialized as an *N*-octet value, where *N* is implementation-defined.

Reference type *RT2* is a *subtype* of data type *RT1* (equivalently, *RT1* is a *supertype* of *RT2*) if and only if *RT1* is a reference type and the referenced type of *RT2* is a subtype of the referenced type of *RT1*.

Every value in a reference type *RT* is a value in every supertype of *RT*. A value *V* in type *RT* has exactly one most specific type *MST* such that *MST* is a subtype of *RT* and *V* is not a value in any proper subtype of *MST*.

A reference type has a *user-defined representation* if its referenced type is defined by a <user-defined type definition> that specifies <user-defined representation>. A reference type has a *derived representation* if its referenced type is defined by a <user-defined type definition> that specifies <derived representation>. A reference type has a *system-defined representation* if it does not have a user-defined representation or a derived representation.

4.9.2 Operations involving references

An operation is provided that takes a REF value and returns the value that is held in a column of the site identified by the REF value (see Subclause 6.20, "<dereference operation>"). If the REF value identifies no site, perhaps because a site it once identified has been destroyed, then the null value is returned.

An operation is provided that takes a REF value and returns a value of the referenced type; that value is constructed from the values of the columns of the site identified by that REF value (see Subclause 6.22, "<reference resolution>"). An operation is also provided that takes a REF value and returns a value acquired from invoking an SQL-invoked method on a value of the referenced type; that value is constructed from the values of the columns of the site identified by that REF value (see Subclause 6.21, "<method reference>").

4.10 Collection types

4.10.1 Introduction to collection types

A *collection* is a composite value comprising zero or more *elements*, each a value of some data type *DT*. If the elements of some collection *C* are values of *DT*, then *C* is said to be a collection of *DT*. The number of elements in *C* is the *cardinality* of *C*. The term "element" is not further defined in this part of ISO/IEC 9075. The term "collection" is generic, encompassing various kinds of collection in connection with each of which, individually, this part of ISO/IEC 9075 defines primitive type constructors and operators. This part of ISO/IEC 9075 supports two kinds of collection types, arrays and multisets.

A specific <collection type> *CT* is a <data type> specified by pairing a keyword *KC* (either ARRAY or MULTISET) with a specific data type *EDT*. In addition, a maximum cardinality may optionally be specified for arrays. Every element of every possible value of *CT* is a value of *EDT* and is permitted to be, more specifically, of some subtype of *EDT*. *EDT* is termed the *element type* of *CT*. *KC* specifies the kind of collection, such as ARRAY or MULTISET, that every value of *CT* is, and thus determines the operators that are available for operating on or returning values of *CT*.

Let *EDTN* be the type designator of *EDT*. The type designator of *CT* is *EDTN KC*.

A *collection type descriptor* describes a collection type. The collection type descriptor for *CT* includes:

— The type designator of *CT*.

— The descriptor of the element type of *CT*.

— An indication of the kind of the collection of *CT*: ARRAY or MULTISET.

— If *CT* is an array type, the maximum number of elements of *CT*.

Collection type *CT2* is a subtype of data type *CT1* (equivalently, *CT1* is a supertype of *CT2*) if and only if *CT1* is the same kind of collection as *CT2* and the element type of *CT2* is a subtype of the element type of *CT1*.

4.10.2 Arrays

An *array* is a collection *A* in which each element is associated with exactly one ordinal position in *A*. If *n* is the cardinality of *A*, then the ordinal position *p* of an element is an integer in the range 1 (one) $\leq p \leq n$. If *EDT* is the element type of *A*, then *A* can thus be considered as a function of the integers in the range 1 (one) to *n* into *EDT*.

An array site *AS* has a maximum cardinality *m*. The cardinality *n* of an array occupying *AS* is constrained not to exceed *m*.

An *array type* is a <collection type>. If *AT* is some array type with element type *EDT*, then every value of *AT* is an array of *EDT*.

Let *A1* and *A2* be arrays of *EDT*. *A1* and *A2* are identical if and only if *A1* and *A2* have the same cardinality *n* and if, for all *i* in the range 1 (one) ≤ *i* ≤ *n*, the element at ordinal position *i* in *A1* is identical to the element at ordinal position *i* in *A2*.

Let *n1* be the cardinality of *A1* and let *n2* be the cardinality of *A2*. *A1* is a *subarray* of *A2* if and only if there exists some *j* in the range 0 (zero) ≤ *j* < *n2* such that, for every *i* in the range 1 (one) ≤ *i* ≤ *n1*, the element at ordinal position *i* in *A1* is the same as the element at ordinal position *i*+*j* in *A2*.

4.10.3 Multisets

A multiset is an unordered collection. Since a multiset is unordered, there is no ordinal position to reference individual elements of a multiset.

A multiset type is a <collection type>. If *MT* is some multiset type with element type *EDT*, then every value of *MT* is a multiset of *EDT*.

Let *M1* and *M2* be multisets of *EDT*. *M1* and *M2* are identical if and only if *M1* and *M2* have the same cardinality *n*, and for each element *x* in *M1*, the number of elements of *M1* that are identical to *x*, including *x* itself, equals the number of elements of *M2* that are identical to *x*.

Let *n1* be the cardinality of *M1* and let *n2* be the cardinality of *M2*. *M1* is a submultiset of *M2* if, for each element *x* of *M1*, the number of elements of *M1* that are not distinct from *x*, including *x* itself, is less than or equal to the number of elements of *M2* that are not distinct from *x*.

4.10.4 Collection comparison and assignment

Two collections are comparable if and only if they are of the same kind of collection (ARRAY or MULTISET) and their element types are comparable.

A value of collection type *CT1* is assignable to a site of collection type *CT2* if and only if *CT1* is the same kind of collection (ARRAY or MULTISET) as *CT2* and the element type of *CT1* is assignable to the element type of *CT2*.

The array types have a defined *element order*. Comparisons are defined in terms of the element order of the arrays. The element order defines the pairs of corresponding elements from the arrays being compared. The element order of an array is implicitly defined by the ordinal position of its elements.

In the case of comparison of two arrays *C* and *D*, the elements are compared pairwise in element order. *C* = *D* is *True* if and only if *C* and *D* have the same cardinality and every pair of elements are equal.

Two multisets *C* and *D* of comparable element types are equal if they have the same cardinality *N* and there exist an enumeration CE_j, 1 (one) ≤ *j* ≤ *N* of the elements of *C* and an enumeration DE_j, 1 (one) ≤ *j* ≤ *N* of the elements of *D* such that for all *j*, $CE_j = DE_j$.

4.10.5 Operations involving arrays

4.10.5.1 Operators that operate on array values and return array elements

<array element reference> is an operation that returns the array element in the specified position within the array.

4.10.5.2 Operators that operate on array values and return array values

<array concatenation> is an operation that returns the array value made by joining its array value operands in the order given.

4.10.6 Operations involving multisets

4.10.6.1 Operators that operate on multisets and return multiset elements

<multiset element reference> is an operation that returns the value of the element of a multiset, if the multiset has only one element.

4.10.6.2 Operators that operate on multisets and return multisets

<multiset set function> is an operation that returns the multiset obtained by removing duplicates from a multiset.

MULTISET UNION is an operator that computes the union of two multisets. There are two variants, specified using ALL or DISTINCT, to either retain duplicates or remove duplicates.

MULTISET INTERSECT is an operator that computes the intersection of two multisets. There are two variants, ALL and DISTINCT. The variant specified by ALL places in the result as many instances of each value as the minimum number of instances of that value in either operand. The variant specified by DISTINCT removes duplicates from the result.

MULTISET EXCEPT is an operator that computes the multiset difference of two multisets. There are two variants, ALL and DISTINCT. The variant specified by ALL places in the result a number of instances of a value, equal to the number of instances of the value in the first operand minus the number of instances of the value in the second operand. The variant specified by DISTINCT removes duplicates from the result.

4.11 Data conversions

Implicit type conversion can occur in expressions, fetch operations, single row select operations, inserts, deletes, and updates. Explicit type conversions can be specified by the use of the CAST operator.

Explicit data conversions can be specified by a *CAST operator*. A CAST operator defines how values of a source data type are converted into a value of a target data type according to the Syntax Rules and General Rules of Subclause 6.12, "<cast specification>". Data conversions between predefined data types and between constructed types are defined by the rules of this part of ISO/IEC 9075. Data conversions between a user-defined type and another data type are defined by a user-defined cast.

A user-defined cast identifies an SQL-invoked function, called the *cast function*, that has one SQL parameter whose declared type is the same as the source data type and a result data type that is the target data type. A cast function may optionally be specified to be implicitly invoked whenever values are assigned to targets of its result data type. Such a cast function is called an *implicitly invocable* cast function.

A user-defined cast is defined by a <user-defined cast definition>. A user-defined cast has a user-defined cast descriptor that includes:

— The name of the source data type.

— The name of the target data type.

— The specific name of the SQL-invoked function that is the cast function.

— An indication as to whether the cast function is implicitly invocable.

When a value *V* of declared type *TV* is assigned to a target *T* of declared type *TT*, a user-defined cast function *UDCF* is said to be an *appropriate user-defined cast function* if and only if all of the following are true:

— The descriptor of *UDCF* indicates that *UDCF* is implicitly invocable.

— The type designator of the declared type *DTP* of the only SQL parameter *P* of *UDCF* is in the type precedence list of *TV*.

— The result data type of *UDCF* is *TT*.

— No other user-defined cast function *UDCQ* with an SQL parameter *Q* with declared type *TQ* that precedes *DTP* in the type precedence list of *TV* is an appropriate user-defined cast function to assign *V* to *T*.

An SQL procedure statement *S* is said to be *dependent on* an appropriate user-defined cast function *UDCF* if and only if all of the following are true:

— *S* is a <select statement: single row>, <insert statement>, <update statement: positioned>, <update statement: searched>, or <merge statement>.

— *UDCF* is invoked during a store or retrieval assignment operation that is executed during the execution of *S* and *UDCF* is not executed during the invocation of an SQL-invoked function that is invoked during the execution of *S*.

4.12 Domains

A domain is a set of permissible values. A domain is defined in a schema and is identified by a <domain name>. The purpose of a domain is to constrain the set of valid values that can be stored in a column of a base table by various operations.

A domain definition specifies a data type. It may also specify a <domain constraint> that further restricts the valid values of the domain and a <default clause> that specifies the value to be used in the absence of an explicitly specified value or column default.

A domain is described by a domain descriptor. A domain descriptor includes:

— The name of the domain.

— The data type descriptor of the data type of the domain.

— The value of <default option>, if any, of the domain.

— The domain constraint descriptors of the domain constraints, if any, of the domain.

4.13 Columns, fields, and attributes

The terms *column*, *field*, and *attribute* refer to structural components of tables, row types, and structured types, respectively, in analogous fashion. As the structure of a table consists of one or more columns, so does the structure of a row type consist of one or more fields and that of a structured type one or more attributes. Every structural element, whether a column, a field, or an attribute, is primarily a name paired with a declared type. The elements of a structure are ordered. Elements in different positions in the same structure can have the same declared type but not the same name. Although the elements of a structure are distinguished from each other by name, in some circumstances the compatibility of two structures (for the purpose at hand) is determined solely by considering the declared types of each pair of elements at the same ordinal position.

A table (see Subclause 4.14, "Tables") is defined on one or more columns and consists of zero or more rows. A column has a name and a declared type. Each row in a table has exactly one value for each column. Each value in a row is a value in the declared type of the column.

NOTE 21 — The declared type includes the null value and values in proper subtypes of the declared type.

Every column has a *nullability characteristic* that indicates whether the value from that column can be the null value. A nullability characteristic is either *known not nullable* or *possibly nullable*.

Let *C* be a column of a base table *T*. *C* is *known not nullable* if and only if at least one of the following is true:

— There exists at least one constraint *NNC* that is not deferrable and that simply contains a <search condition> that is a <boolean value expression> that is a known-not-null condition for *C*.

— *C* is based on a domain that has a domain constraint that is not deferrable and that simply contains a <search condition> that is a <boolean value expression> that is a known-not-null condition for VALUE.

— *C* is a unique column of a nondeferrable unique constraint that is a PRIMARY KEY.

— The SQL-implementation is able to deduce that the <search condition> "*C* IS NULL" can never be true when applied to a row in *T* through some additional implementation-defined rule or rules.

The nullability characteristic of a column of a derived table is defined by the the Syntax Rules of Subclause 7.7, "<joined table>", Subclause 7.12, "<query specification>", and Subclause 7.13, "<query expression>".

A column *C* is described by a column descriptor. A column descriptor includes:

— The name of the column.

— Whether the name of the column is an implementation-dependent name.

— If the column is based on a domain, then the name of that domain; otherwise, the data type descriptor of the declared type of *C*.

— The value of <default option>, if any, of *C*.

— The nullability characteristic of *C*.

— The ordinal position of *C* within the table that contains it.

— An indication of whether *C* is a self-referencing column of a base table or not.

— An indication of whether *C* is an identity column or not.

— If *C* is an identity column, then an indication of whether values are always generated or generated by default.

— If *C* is an identity column, then the *start value* of *C*.

— If *C* is an identity column, then the descriptor of the internal sequence generator for *C*.

NOTE 22 — Identity columns and the meaning of "start value" are described in Subclause 4.14.7, "Identity columns".

— If *C* is a generated column, then the generation expression of *C*.

NOTE 23 — Generated columns and the meaning of "generation expression" are described in Subclause 4.14.8, "Base columns and generated columns".

An attribute *A* is described by an attribute descriptor. An attribute descriptor includes:

— The name of the attribute.

— The data type descriptor of the declared type of *A*.

— The ordinal position of *A* within the structured type that contains it.

— The value of the implicit or explicit <attribute default> of *A*.

— The name of the structured type defined by the <user-defined type definition> that defines *A*.

A field *F* is described by a field descriptor. A field descriptor includes:

— The name of the field.

— The data type descriptor of the declared type of *F*.

— The ordinal position of *F* within the row type that simply contains it.

4.14 Tables

4.14.1 Introduction to tables

A table is a collection of rows having one or more columns. A row is a value of a row type. Every row of the same table has the same row type. The value of the *i*-th field of every row in a table is the value of the *i*-th column of that row in the table. The row is the smallest unit of data that can be inserted into a table and deleted from a table.

A table *T2* is *part of* a column *C* of a table *T1* if setting the value of *T1.C* to a null value (ignoring any constraints or triggers defined on *T1* or *T1.C*) would cause *T2* to disappear.

The most specific type of a row is a row type. All rows of a table are of the same row type and this is called the *row type* of that table.

The degree of a table, and the degree of each of its rows, is the number of columns of that table. The number of rows in a table is its cardinality. A table whose cardinality is 0 (zero) is said to be *empty*.

4.14.2 Types of tables

A table is either a base table, a derived table, or a transient table. A base table is either a persistent base table, a global temporary table, a created local temporary table, or a declared local temporary table.

A persistent base table is a named table defined by a <table definition> that does not specify TEMPORARY.

A derived table is a table derived directly or indirectly from one or more other tables by the evaluation of a <query expression> whose result has an element type that is a row type. The values of a derived table are derived from the values of the underlying tables when the <query expression> is evaluated.

A viewed table is a named derived table defined by a <view definition>. A viewed table is sometimes called a *view*.

A transient table is a named table that may come into existence implicitly during the evaluation of a <query expression> or the execution of a trigger. A transient table is identified by a <query name> if it arises during the evaluation of a <query expression>, or by a <transition table name> if it arises during the execution of a trigger. Such tables exist only for the duration of the executing SQL-statement containing the <query expression> or for the duration of the executing trigger.

A table is either *updatable* or *not updatable*. An updatable table has at least one *updatable column*. If every column of table *T* is updatable, then *T* is *fully updatable*. An updatable table that is not fully updatable is *partially updatable*. All base tables are fully updatable. Derived tables and transient tables are either updatable or not updatable. The operations of update and delete are permitted for updatable tables, subject to constraining Access Rules. Some updatable tables, including all base tables whose row type is not derived from a user-defined type that is not instantiable, are also *insertable-into*, in which case the operation of insert is also permitted, again subject to Access Rules.

A *grouped table* is a set of groups derived during the evaluation of a <group by clause>. A group *G* is a collection of rows in which, for every grouping column *GC*, if the value of *GC* in some row is not distinct from *GV*, then

the value of *GC* in every row is *GV*; moreover, if *R1* is a row in group *G1* of grouped table *GT* and *R2* is a row in *GT* such that for every grouping column *GC* the value of *GC* in *R1* is not distinct from the value of *GC* in *R2*, then *R2* is in *G1*. Every row in *GT* is in exactly one group. A group may be considered as a table. Set functions operate on groups.

A global temporary table is a named table defined by a <table definition> that specifies GLOBAL TEMPORARY. A created local temporary table is a named table defined by a <table definition> that specifies LOCAL TEMPORARY. Global and created local temporary tables are effectively materialized only when referenced in an SQL-session. Every SQL-client module in every SQL-session that references a created local temporary table causes a distinct instance of that created local temporary table to be materialized. That is, the contents of a global temporary table or a created local temporary table cannot be shared between SQL-sessions.

In addition, the contents of a created local temporary table cannot be shared between SQL-client modules of a single SQL-session. The definition of a global temporary table or a created local temporary table appears in a schema. In SQL language, the name and the scope of the name of a global temporary table or a created local temporary table are indistinguishable from those of a persistent base table. However, because global temporary table contents are distinct within SQL-sessions, and created local temporary tables are distinct within SQL-client modules within SQL-sessions, the *effective* <schema name> of the schema in which the global temporary table or the created local temporary table is instantiated is an implementation-dependent <schema name> that may be thought of as having been effectively derived from the <schema name> of the schema in which the global temporary table or created local temporary table is defined and the implementation-dependent SQL-session identifier associated with the SQL-session.

In addition, the *effective* <schema name> of the schema in which the created local temporary table is instantiated may be thought of as being further qualified by a unique implementation-dependent name associated with the SQL-client module in which the created local temporary table is referenced.

A module local temporary table is a named table defined by a <temporary table declaration> in an SQL-client module. A module local temporary table is effectively materialized the first time it is referenced in an SQL-session, and it persists for that SQL-session.

A declared local temporary table may be declared in an SQL-client module.

A declared local temporary table is a module local temporary table. A declared local temporary table is accessible only by externally-invoked procedures in the SQL-client module that contains the <temporary table declaration> that declares the declared local temporary table. The effective <schema name> of the <schema qualified name> of the declared local temporary table may be thought of as the implementation-dependent SQL-session identifier associated with the SQL-session and a unique implementation-dependent name associated with the <SQL-client module definition> that contains the <temporary table declaration>.

All references to a declared local temporary table are prefixed by "MODULE.".

The materialization of a temporary table does not persist beyond the end of the SQL-session in which the table was materialized. Temporary tables are effectively empty at the start of an SQL-session.

4.14.3 Table descriptors

A table is described by a table descriptor. A table descriptor is either a base table descriptor, a view descriptor, or a derived table descriptor (for a derived table that is not a view).

Every table descriptor includes:

— The column descriptor of each column in the table.

— The name, if any, of the structured type, if any, associated with the table.

— An indication of whether the table is insertable-into or not.

— An indication of whether the table is a referenceable table or not, and an indication of whether the self-referencing column is a system-generated, a user-generated, or a derived self-referencing column.

— A list, possibly empty, of the names of its direct supertables.

— A list, possibly empty, of the names of its direct subtables.

A transient table descriptor describes a transient table. In addition to the components of every table descriptor, a transient table descriptor includes:

— If the transient table is defined by a <with list element> contained in a <query expression>, then the <query name>. If the transient table is defined by a <trigger definition> then the <transition table name>.

A base table descriptor describes a base table. In addition to the components of every table descriptor, a base table descriptor includes:

— The name of the base table.

— An indication of whether the table is a persistent base table, a global temporary table, a created local temporary table, or a declared local temporary table.

— If the base table is a global temporary table, a created local temporary table, or a declared local temporary table, then an indication of whether ON COMMIT PRESERVE ROWS was specified or ON COMMIT DELETE ROWS was specified or implied.

— The descriptor of each table constraint specified for the table.

— A non-empty set of functional dependencies, according to the rules given in Subclause 4.18, "Functional dependencies".

— A non-empty set of candidate keys, according to the rules of Subclause 4.19, "Candidate keys".

— A preferred candidate key, which may or may not be additionally designated the primary key, according to the Rules in Subclause 4.18, "Functional dependencies".

A derived table descriptor describes a derived table. In addition to the components of every table descriptor, a derived table descriptor includes:

— The <query expression> that defines how the table is to be derived.

— An indication of whether the derived table is updatable or not.

A view descriptor describes a view. In addition to the components of a derived table descriptor, a view descriptor includes:

— The name of the view.

— An indication of whether the view has the CHECK OPTION; if so, whether it is to be applied as CASCADED or LOCAL.

— The original <query expression> of the view.

4.14.4 Relationships between tables

The terms *simply underlying table*, *underlying table*, *leaf underlying table*, *generally underlying table*, and *leaf generally underlying table* define a relationship between a derived table or cursor and other tables.

The *simply underlying table*s of derived tables and cursors are defined in Subclause 7.12, "<query specification>", Subclause 7.13, "<query expression>", and Subclause 14.1, "<declare cursor>". A <table or query name> has no simply underlying tables.

The *underlying table*s of a derived table or cursor are the simply underlying tables of the derived table or cursor and the underlying tables of the simply underlying tables of the derived table or cursor.

The *leaf underlying table*s of a derived table or cursor are the underlying tables of the derived table or cursor that do not themselves have any underlying tables.

The *generally underlying table*s of a derived table or cursor are the underlying tables of the derived table or cursor and, for each underlying table of the derived table or cursor that is a <table or query name> *TORQN*, the generally underlying tables of *TORQN*, which are defined as follows:

— If *TORQN* identifies a base table or if *TORQN* is a <transition table name>, then *TORQN* has no generally underlying tables.

— If *TORQN* is a <query name>, then the generally underlying tables of *TORQN* are the <query expression body> *QEB* of the <with list element> identified by *TORQN* and the generally underlying tables of *QEB*.

— If *TORQN* identifies a view *V*, then the generally underlying tables of *TORQN* are the <query expression> *QEV* included in the view descriptor of *V* and the generally underlying tables of *QEV*.

The *leaf generally underlying table*s of a derived table or cursor are the generally underlying tables of the derived table or cursor that do not themselves have any generally underlying tables.

4.14.5 Referenceable tables, subtables, and supertables

A table *RT* whose row type is derived from a structured type *ST* is called a *typed table*. Only a base table or a view can be a typed table. A typed table has columns corresponding, in name and declared type, to every attribute of *ST* and one other column *REFC* that is the self-referencing column of *RT*; let *REFCN* be the <column name> of *REFC*. The declared type of *REFC* is necessarily REF(*ST*) and the nullability characteristic of *REFC* is *known not nullable*. If *RT* is a base table, then the table constraint "UNIQUE(*REFCN*)" is implicit in the definition of *RT*. A typed table is called a *referenceable table*. A self-referencing column cannot be updated. Its value is determined during the insertion of a row into the referenceable table. The value of a system-generated self-referencing column and a derived self-referencing column is automatically generated when the row is inserted into the referenceable table. The value of a user-generated self-referencing column is supplied as part of the candidate row to be inserted into the referenceable table.

A table T_a is a *direct subtable* of another table T_b if and only if the <table name> of T_b is contained in the <subtable clause> contained in the <table definition> or <view definition> of T_a. Both T_a and T_b shall be created on a structured type and the structured type of T_a shall be a direct subtype of the structured type of T_b.

A table T_a is a *subtable* of a table T_b if and only if any of the following are true:

1) T_a and T_b are the same named table.

2) T_a is a direct subtable of T_b.

3) There is a table T_c such that T_a is a direct subtable of T_c and T_c is a subtable of T_b.

A table T is considered to be one of its own subtables. Subtables of T other than T itself are called its *proper subtables*. A table shall not have itself as a proper subtable.

A table T_b is called a *supertable* of a table T_a if T_a is a subtable of T_b. If T_a is a direct subtable of T_b, then T_b is called a *direct supertable* of T_a. A table that is not a subtable of any other table is called a *maximal supertable*.

Let T_a be a maximal supertable and T be a subtable of T_a. The set of all subtables of T_a (which includes T_a itself) is called the *subtable family* of T or (equivalently) of T_a. Every subtable family has exactly one maximal supertable.

A *leaf table* is a table that does not have any proper subtables.

Those columns of a subtable T_a of a structured type ST_a that correspond to the inherited attributes of ST_a are called *inherited columns*. Those columns of T_a that correspond to the originally-defined attributes of ST_a are called *originally-defined columns*.

Let *TB* be a subtable of *TA*. Let *SLA* be the <value expression> sequence implied by the <select list> "*" in the <query specification> "SELECT * FROM *TA*". For every row *RB* in the value of *TB* there exists exactly one row *RA* in the value of *TA* such that *RA* is the result of the <row subquery> "SELECT *SLA* FROM VALUES *RRB*", where *RRB* is some <row value constructor> whose value is *RB*. *RA* is said to be the *superrow* in *TA* of *RB* and *RB* is said to be the *subrow* in *TB* of *RA*. If *TA* is a base table, then the one-to-one correspondence between superrows and subrows is guaranteed by the requirement for a unique constraint to be specified for some supertable of *TA*. If *TA* is a view, then such one-to-one correspondence is guaranteed by the requirement for a unique constraint to be specified on the leaf generally underlying table of *TA*.

Users shall have the UNDER privilege on a table before they can use the table in a subtable definition. A table can have more than one proper subtable. Similarly, a table can have more than one proper supertable.

4.14.6 Operations involving tables

Table values are operated on and returned by <query expression>s. The syntax of <query expression> includes various internal operators that operate on table values and return table values. In particular, every <query expression> effectively includes at least one <from clause>, which operates on one or more table values and returns a single table value. A table value operated on by a <from clause> is specified by a <table reference>.

An operation involving a table *T* may define a *range variable RV* that ranges over rows of *T*, referencing each row in turn in an implementation-dependent order. Thus, each reference to *RV* references exactly one row of *T*. *T* is said to be the *table associated with RV*.

In a <table reference>, ONLY can be specified to exclude from the result rows that have subrows in proper subtables of the referenced table.

In a <table reference>, <sample clause> can be specified to return a subset of result rows depending on the <sample method> and <sample percentage>. If the <sample clause> contains <repeatable clause>, then repeated executions of that <table reference> return a result table with identical rows for a given <repeat argument>, provided certain implementation-defined conditions are satisfied.

A <table reference> that satisfies certain properties specified in this international standard can be used to designate an *updatable table*. Certain table updating operations, specified by SQL-data change statements, are available in connection with updatable tables. The value of an updatable table *T* is determined by the result of the mostly recently executed SQL-data change statement (see Subclause 4.33.2, "SQL-statements classified by function") operating on *T*. An SQL-data change statement on table *T* has a *primary effect* (on *T* itself) and zero or more *secondary effects* (not necessarily on *T*).

The primary effect of a <delete statement: positioned> on a table *T* is to delete exactly one specified row from *T*. The primary effect of a <delete statement: searched> on a table *T* is to delete zero or more rows from *T*.

The primary effect of an <update statement: positioned> on a table *T* is to replace exactly one specified row in *T* with some specified row. The primary effect of an <update statement: searched> on a table *T* is to replace zero or more rows in *T*.

If a table *T*, as well as being updatable, is insertable-into, then rows can be inserted into it. The primary effect of an <insert statement> on *T* is to insert into *T* each of the zero or more rows contained in a specified table. The primary effect of a <merge statement> on *T* is to replace zero or more rows in *T* with specified rows and/or to insert into *T* zero or more specified rows, depending on the result of a <search condition> and on whether one or both of <merge when matched clause> and <merge when not matched clause> are specified.

Each of the table updating operations, when applied to a table *T*, can have various secondary effects. Such secondary effects can include alteration or reversal of the primary effect. Secondary effects might arise from the existence of:

— Underlying tables of *T*, other than *T* itself, whose values might be subject to secondary effects.

— Updatable views whose <view definition>s do not specify WITH CASCADED CHECK OPTION.

— Cascaded operations specified in connection with integrity constraints involving underlying tables of *T*, which might result in secondary effects on tables referenced by such constraints.

— Proper subtables and proper supertables of *T*, whose values might be affected by updating operations on *T*.

— Triggers specified for underlying tables of *T*, which might specify table updating operations on updatable tables other than *T*.

4.14.7 Identity columns

The columns of a base table *BT* can optionally include not more than one *identity column*. The declared type of an identity column is either an exact numeric type with scale 0 (zero), INTEGER for example, or a distinct type whose source type is an exact numeric type with scale 0 (zero). An identity column has a *start value*, an *increment*, a *maximum value*, a *minimum value*, and a *cycle option*. An identity column is associated with an internal sequence generator *SG*. Let *IC* be the identity column of *BT*. When a row *R* is presented for insertion into *BT*, if *R* does not contain a column corresponding to *IC*, then the value *V* for *IC* in the row inserted into *BT* is obtained by applying the General Rules of Subclause 9.21, "Generation of the next value of a sequence generator", to *SG*. The definition of an identity column may specify GENERATED ALWAYS or GENERATED BY DEFAULT.

NOTE 24 — "Start value", "increment", "maximum value", "minimum value", and "cycle option" are defined in Subclause 4.21, "Sequence generators".

NOTE 25 — The notion of an internal sequence generator being associated with an identity column is used only for definitional purposes in this International Standard.

4.14.8 Base columns and generated columns

A column of a base table is either a *base column* or a *generated column*. A base column is one that is not a generated column. A generated column is one whose values are determined by evaluation of a *generation expression*, a <value expression> whose declared type is by implication that of the column. A generation expression can reference base columns of the base table to which it belongs but cannot otherwise access SQL-data. Thus, the value of the field corresponding to a generated column in row *R* is determined by the values of zero or more other fields of *R*.

A generated column *GC* depends on each column that is referenced by a <column reference> in its generation expression, and each such referenced column is a *parametric column* of *GC*.

4.14.9 Windowed tables

A *windowed table* is a table together with one or more windows. A *window* is a transient data structure associated with a <table expression>. A window is defined explicitly by a <window definition> or implicitly by an <in-line window specification>. Implicitly defined windows have an implementation-dependent window name. A window is used to specify window partitions and window frames, which are collections of rows used in the definition of <window function>s.

Every window defines a *window partitioning* of the rows of the <table expression>. The window partitioning is specified by a list of columns. Window partitioning is similar to forming groups of a grouped table. However, unlike grouped tables, each row is retained in the result of the <table expression>. The *window partition* of a row *R* is the collection of rows *R2* that are not distinct from *R*, for all columns enumerated in the window partitioning clause. The window partitioning clause is optional; if omitted, there is a single window partition consisting of all the rows in the result.

If a <table expression> is grouped and also has a window, then there is a syntactic transformation that segregates the grouping into a <derived table>, so that the window partitions consist of rows of the <derived table> rather than groups of rows.

A window may define a *window ordering* of rows within each window partition defined by the window. The window ordering of rows within window partitions is specified by a list of <value expression>s, followed by ASC (for ascending order) or DESC (for descending order). In addition, NULLS FIRST or NULLS LAST may be specified, to indicate whether a null value should appear before or after all non-null values in the ordered sequence of each <value expression>.

Optionally, a window may define a *window frame* for each row *R*. A window frame is always defined relative to the current row. A window frame is specified by up to four syntactic elements:

— The choice of RANGE, to indicate a logical definition of the window frame by offsetting forward or backward from the current row by an increment or decrement to the sort key; or ROWS, to indicate a physical definition of the window frame, by counting rows forward or backward from the current row.

— A starting row, which may be the first row of the window partition of *R*, the current row, or some row determined by a logical or physical offset from the current row.

— An ending row, which may be the last row of the window partition of *R*, the current row, or some row determined by a logical or physical offset from the current row.

— A <window frame exclusion>, indicating whether to exclude the current row and/or its peers (if not already excluded by being prior to the starting row or after the ending row).

A window is described by a *window structure descriptor*, including:

— The window name.

— Optionally, the ordering window name—that is, the name of another window, called the *ordering window*, that is used to define the partitioning and ordering of the present window.

— The window partitioning clause—that is, a <window partition clause> if any is specified in either the present <window specification> or in the window descriptor of the ordering window.

— The window ordering clause—that is, a <window order clause> if any is specified in either the present <window specification> or in the window descriptor of the ordering window.

— The window framing clause—that is, a <window frame clause>, if any.

In general, two <window function>s are computed independently, each one performing its own sort of its data, even if they use the same data and the same <sort specification list>. Since sorts may specify partial orderings, the computation of <window function>s is inevitably non-deterministic to the extent that the ordering is not total. Nevertheless, the user may desire that two <window function>s be computed using the same ordering, so that, for example, two moving aggregates move through the rows of a partition in precisely the same order. Two <window function>s are computed using the same (possibly non-deterministic) window ordering of the rows if any of the following are true:

— The <window function>s identify the same window structure descriptor.

— The <window function>s' window structure descriptors have window partitioning clauses that enumerate the same number of column references, and those column references are pairwise equivalent in their order of occurrence; and their window structure descriptors have window ordering clauses with the same number of <sort key>s, and those <sort key>s are all column references, and those column references are pairwise

equivalent in their order of occurrence, and the <sort specification>s pairwise specify or imply <collate clause>s that specify equivalent <collation name>s, the same <ordering specification> (ASC or DESC), and the same <null ordering> (NULLS FIRST or NULLS LAST).

— The window structure descriptor of one <window function> is the ordering window of the other <window function>, or both window structure descriptors identify the same ordering window.

4.15 Data analysis operations (involving tables)

4.15.1 Introduction to data analysis operations

A data analysis function is a function that returns a value derived from a number of rows in the result of a <table expression>. A data analysis function may only be invoked as part of a <query specification> or <select statement: single row>, and then only in certain contexts, identified below. A data analysis function is one of:

— A group function, which is invoked on a grouped table and computes a grouping operation or an aggregate function from a group of the grouped table.

— A window function, which is invoked on a windowed table and computes a rank, row number or window aggregate function.

4.15.2 Group functions

A group function may only appear in the <select list>, <having clause> or <window clause> of a <query specification> or <select statement: single row>, or in the <order by clause> of a cursor that is a simple table query.

A group function is one of:

— The *grouping operation*.

— A *group aggregate function*.

The grouping operation is of the form `GROUPING (<column reference>)`. The result of such an invocation is 1 (one) in the case of a row whose values are the results of aggregation over that <column reference> during the execution of a grouped query containing CUBE, ROLLUP, or GROUPING SET, and 0 (zero) otherwise.

4.15.3 Window functions

A window function is a function whose result for a given row is derived from the window frame of that row as defined by a window structure descriptor of a windowed table. Window functions may only appear in the <select list> of a <query specification> or <select statement: single row>, or the <order by clause> of a simple table query.

A window function is one of:

— A rank function.

— A distribution function.

— The row number function.

— A window aggregate function.

The rank functions compute the ordinal rank of a row *R* within the window partition of *R* as defined by a window structure descriptor, according to the window ordering of those rows, also specified by the same window structure descriptor. Rows that are not distinct with respect to the window ordering within their window partition are assigned the same rank. There are two variants, indicated by the keywords RANK and DENSE_RANK.

— If RANK is specified, then the rank of row *R* is defined as 1 (one) plus the number of rows that precede *R* and are not peers of *R*.

NOTE 26 — This implies that if two or more rows are not distinct with respect to the window ordering, then there will be one or more gaps in the sequential rank numbering.

— If DENSE_RANK is specified, then the rank of row *R* is defined as the number of rows preceding and including *R* that are distinct with respect to the window ordering.

NOTE 27 — This implies that there are no gaps in the sequential rank numbering of rows in each window partition.

The distribution functions compute a relative rank of a row *R* within the window partition of *R* defined by a window structure descriptor, expressed as an approximate numeric ratio between 0.0 and 1.0. There are two variants, indicated by the keywords PERCENT_RANK and CUME_DIST.

— If PERCENT_RANK is specified, then the relative rank of a row *R* is defined as $(RK-1)/(NR-1)$, where *RK* is defined to be the RANK of *R* and *NR* is defined to be the number of rows in the window partition of *R*.

— If CUME_DIST is specified, then the relative rank of a row *R* is defined as *NP*/*NR*, where *NP* is defined to be the number of rows preceding or peer with *R* in the window ordering of the window partition of *R* and *NR* is defined to be the number of rows in the window partition of *R*.

The ROW_NUMBER function computes the sequential row number, starting with 1 (one) for the first row, of the row within its window partition according to the window ordering of the window.

The window aggregate functions compute an aggregate value (COUNT, SUM, AVG, *etc.*), the same as a group aggregate function, except that the computation aggregates over the window frame of a row rather than over a group of a grouped table. The hypothetical set functions are not permitted as window aggregate functions.

4.15.4 Aggregate functions

An aggregate function is a function whose result is derived from an aggregation of rows defined by one of:

— The grouping of a grouped table, in which case the aggregate function is a group aggregate function, or set function, and for each group there is one aggregation, which includes every row in the group.

— The window frame of a row *R* of a windowed table relative to a particular window structure descriptor, in which case the aggregate function is a window aggregate function, and the aggregation consists of every row in the window frame of *R*, as defined by the window structure descriptor.

Optionally, the collection of rows in an aggregation may be filtered, retaining only those rows that satisfy a <search condition> that is specified by a <filter clause>.

The result of the aggregate function COUNT (*) is the number of rows in the aggregation.

Every other aggregate function may be classified as a *unary group aggregate function*, a *binary group aggregate functions*, an *inverse distribution*, or a *hypothetical set function*.

Every unary aggregate function takes an arbitrary <value expression> as the argument; most unary aggregate functions can optionally be qualified with either DISTINCT or ALL. Of the rows in the aggregation, the following do not qualify:

— If DISTINCT is specified, then redundant duplicates.

— Every row in which the <value expression> evaluates to the null value.

If no row qualifies, then the result of COUNT is 0 (zero), and the result of any other aggregate function is the null value.

Otherwise (*i.e.*, at least one row qualifies), the result of the aggregate function is:

— If COUNT <value expression> is specified, then the number of rows that qualify.

— If SUM is specified, then the sum of <value expression> evaluated for each row that qualifies.

— If AVG is specified, then the average of <value expression> evaluated for each row that qualifies.

— If MAX is specified, then the maximum value of <value expression> evaluated for each row that qualifies.

— If MIN is specified, then the minimum value of <value expression> evaluated for each row that qualifies.

— If EVERY is specified, then *True* if the <value expression> evaluates to *True* for every row that qualifies; otherwise, *False*.

— If ANY or SOME is specified, then *True* if the <value expression> evaluates to *True* for at least one row remaining in the group; otherwise, *False*.

— If VAR_POP is specified, then the population variance of <value expression> evaluated for each row remaining in the group, defined as the sum of squares of the difference of <value expression> from the mean of <value expression>, divided by the number of rows remaining.

— If VAR_SAMP is specified, then the sample variance of <value expression> evaluated for each row remaining in the group, defined as the sum of squares of the difference of <value expression> from the mean of <value expression>, divided by the number of rows remaining minus 1 (one).

— If STDDEV_POP is specified, then the population standard deviation of <value expression> evaluated for each row remaining in the group, defined as the square root of the population variance.

— If STDDEV_SAMP is specified, then the sample standard deviation of <value expression> evaluated for each row remaining in the group, defined as the square root of the sample variance.

Neither DISTINCT nor ALL are allowed to be specified for VAR_POP, VAR_SAMP, STDDEV_POP, or STDDEV_SAMP; redundant duplicates are not removed when computing these functions.

The binary aggregate functions take a pair of arguments, the <dependent variable expression> and the <independent variable expression>, which are both <numeric value expression>s. Any row in which either argument evaluates to the null value is removed from the group. If there are no rows remaining in the group, then the result of REGR_COUNT is 0 (zero), and the other binary aggregate functions result in the null value. Otherwise, the computation concludes and the result is:

— If REGR_COUNT is specified, then the number of rows remaining in the group.

— If COVAR_POP is specified, then the population covariance, defined as the sum of products of the difference of <independent variable expression> from its mean times the difference of <dependent variable expression> from its mean, divided by the number of rows remaining.

— If COVAR_SAMP is specified, then the sample covariance, defined as the sum of products of the difference of <independent variable expression> from its mean times the difference of <dependent variable expression> from its mean, divided by the number of rows remaining minus 1 (one).

— If CORR is specified, then the correlation coefficient, defined as the ratio of the population covariance divided by the product of the population standard deviation of <independent variable expression> and the population standard deviation of <dependent variable expression>.

— If REGR_R2 is specified, then the square of the correlation coefficient.

— If REGR_SLOPE is specified, then the slope of the least-squares-fit linear equation determined by the (<independent variable expression>, <dependent variable expression>) pairs.

— If REGR_INTERCEPT is specified, then the y-intercept of the least-squares-fit linear equation determined by the (<independent variable expression>, <dependent variable expression>) pairs.

— If REGR_SXX is specified, then the sum of squares of <independent variable expression>.

— If REGR_SYY is specified, then the sum of squares of <dependent variable expression>.

— If REGR_SXY is specified, then the sum of products of <independent variable expression> times <dependent variable expression>.

— If REGR_AVGX is specified, then the average of <independent variable expression>.

— If REGR_AVGY is specified, then the average of <dependent variable expression>.

There are two inverse distribution functions, PERCENTILE_CONT and PERCENTILE_DISC. Both inverse distribution functions specify an argument and an ordering of a value expression. The value of the argument should be between 0 (zero) and 1 (one) inclusive. The value expression is evaluated for each row of the group, nulls are discarded, and the remaining rows are ordered. The computation concludes:

— If PERCENTILE_CONT is specified, by considering the pair of consecutive rows that are indicated by the argument, treated as a fraction of the total number of rows in the group, and interpolating the value of the value expression evaluated for these rows.

— If PERCENTILE_DISC is specified, by treating the group as a window partition of the CUME_DIST window function, using the specified ordering of the value expression as the window ordering, and returning the first value expression whose cumulative distribution value is greater than or equal to the argument.

The hypothetical set functions are related to the window functions RANK, DENSE_RANK, PERCENT_RANK, and CUME_DIST, and use the same names, though with a different syntax. These functions take an argument *A* and an ordering of a value expression *VE*. *VE* is evaluated for all rows of the group. This collection of values is augmented with *A*; the resulting collection is treated as a window partition of the corresponding window function whose window ordering is the ordering of the value expression. The result of the hypothetical set function is the value of the eponymous window function for the hypothetical "row" that contributes *A* to the collection.

4.16 Determinism

In general, an operation is *deterministic* if that operation assuredly computes identical results when repeated with identical input values. For an SQL-invoked routine, the values in the argument list are regarded as the input; otherwise, the SQL-data and the set of privileges by which they are accessed is regarded as the input. Differences in the ordering of rows, as permitted by General Rules that specify implementation-dependent behavior, are not regarded as significant to the question of determinism.

NOTE 28 — Transaction isolation levels have a significant impact on determinism, particularly isolation levels other than SERIALIZABLE. However, this International Standard does not address that impact, particularly because of the difficulty in clearly specifying that impact without appearing to mandate implementation techniques (such as row or page locking) and because different SQL-implementations almost certainly resolve the issue in significantly different ways.

Recognizing that an operation is deterministic is a difficult task, it is in general not mandated by this International Standard. SQL-invoked routines are regarded as deterministic if the routine is declared to be DETERMINISTIC; that is, the SQL-implementation trusts the definer of the SQL-invoked routine to correctly declare that the routine is deterministic. For other operations, this International Standard does not label an operation as deterministic; instead it identifies certain operations as "possibly non-deterministic". Specific definitions can be found in other subclauses relative to <value expression>, <table reference>, <table primary>, <query specification>, <query expression>, and <SQL procedure statement>.

Certain <boolean value expression>s are identified as "retrospectively deterministic". A retrospectively deterministic <boolean value expression> has the property that if it is *True* at one point time, then it is *True* for all later points in time if re-evaluated for the identical SQL-data by an arbitrary user with the identical set of privileges. The precise definition is found in Subclause 6.34, "<boolean value expression>".

4.17 Integrity constraints

4.17.1 Overview of integrity constraints

Integrity constraints, generally referred to simply as constraints, define the valid states of SQL-data by constraining the values in the base tables. A constraint is described by a constraint descriptor. A constraint is either a table constraint, a domain constraint, or an assertion and is described by, respectively, a table constraint descriptor, a domain constraint descriptor, or an assertion descriptor. Every constraint descriptor includes:

— The name of the constraint.

— An indication of whether or not the constraint is deferrable.

— An indication of whether the initial constraint mode is *deferred* or *immediate*.

No integrity constraint shall be defined using a <search condition> that is not retrospectively deterministic.

4.17.2 Checking of constraints

Every constraint is either *deferrable* or *non-deferrable*. Within an SQL-transaction, every constraint has a constraint mode; if a constraint is *non-deferrable*, then its constraint mode is always *immediate*, otherwise it is either *immediate* or *deferred*. Every constraint has an initial constraint mode that specifies the constraint mode for that constraint at the start of each SQL-transaction and immediately after definition of that constraint. If a constraint is *deferrable*, then its constraint mode may be changed (from *immediate* to *deferred*, or from *deferred* to *immediate*) by execution of a <set constraints mode statement>.

The checking of a constraint depends on its constraint mode within the current SQL-transaction. If the constraint mode is *immediate*, then the constraint is effectively checked at the end of each SQL-statement.

NOTE 29 — This includes SQL-statements that are executed as a direct result or an indirect result of executing a different SQL-statement.

If the constraint mode is *deferred*, then the constraint is effectively checked when the constraint mode is changed to *immediate* either explicitly by execution of a <set constraints mode statement>, or implicitly at the end of the current SQL-transaction.

When a constraint is checked other than at the end of an SQL-transaction, if it is not satisfied, then an exception condition is raised and the SQL-statement that caused the constraint to be checked has no effect other than entering the exception information into the first diagnostics area. When a <commit statement> is executed, all constraints are effectively checked and, if any constraint is not satisfied, then an exception condition is raised and the SQL-transaction is terminated by an implicit <rollback statement>.

4.17.3 Table constraints

A table constraint is either a unique constraint, a referential constraint or a table check constraint. A table constraint is described by a table constraint descriptor which is either a unique constraint descriptor, a referential constraint descriptor or a table check constraint descriptor.

Every table constraint specified for base table *T* is implicitly a constraint on every subtable of *T*, by virtue of the fact that every row in a subtable is considered to have a corresponding superrow in every one of its supertables.

A unique constraint is satisfied if and only if no two rows in a table have the same non-null values in the *unique columns*. In addition, if the unique constraint was defined with PRIMARY KEY, then it requires that none of the values in the specified column or columns be a null value.

A unique constraint is described by a unique constraint descriptor. In addition to the components of every table constraint descriptor, a unique constraint descriptor includes:

— An indication of whether it was defined with PRIMARY KEY or UNIQUE.

— The names and positions of the *unique columns* specified in the <unique column list>.

If the table descriptor for base table *T* includes a unique constraint descriptor indicating that the unique constraint was defined with PRIMARY KEY, then the columns of that unique constraint constitute the *primary key* of *T*. A table that has a primary key cannot have a proper supertable.

A referential constraint is described by a referential constraint descriptor. In addition to the components of every table constraint descriptor, a referential constraint descriptor includes:

— A list of the names of the *referencing columns* specified in the <referencing columns>.

— The *referenced table* specified in the <referenced table and columns>.

— A list of the names of the *referenced columns* specified in the <referenced table and columns>.

— The value of the <match type>, if specified, and the <referential triggered action>s, if specified.

NOTE 30 — If MATCH FULL or MATCH PARTIAL is specified for a referential constraint and if the referencing table has only one column specified in <referential constraint definition> for that referential constraint, or if the referencing table has more than one specified column for that <referential constraint definition>, but none of those columns is nullable, then the effect is the same as if no <match type> were specified.

The ordering of the lists of referencing column names and referenced column names is implementation-defined, but shall be such that corresponding column names occupy corresponding positions in each list.

In the case that a table constraint is a referential constraint, the table is referred to as the *referencing table*. The *referenced columns* of a referential constraint shall be the *unique columns* of some unique constraint of the *referenced table*.

A referential constraint is satisfied if one of the following conditions is true, depending on the <match type> specified in the <referential constraint definition>:

— If no <match type> was specified then, for each row *R1* of the *referencing table*, either at least one of the values of the *referencing columns* in *R1* shall be a null value, or the value of each referencing column in *R1* shall be equal to the value of the corresponding *referenced column* in some row of the *referenced table*.

— If MATCH FULL was specified then, for each row *R1* of the *referencing table*, either the value of every *referencing column* in *R1* shall be a null value, or the value of every *referencing column* in *R1* shall not be null and there shall be some row *R2* of the *referenced table* such that the value of each *referencing column* in *R1* is equal to the value of the corresponding *referenced column* in *R2*.

— If MATCH PARTIAL was specified then, for each row *R1* of the *referencing table*, there shall be some row *R2* of the *referenced table* such that the value of each *referencing column* in *R1* is either null or is equal to the value of the corresponding *referenced column* in *R2*.

The referencing table may be the same table as the referenced table.

A table check constraint is described by a table check constraint descriptor. In addition to the components of every table constraint descriptor, a table check constraint descriptor includes:

— The <search condition>.

A table check constraint is satisfied if and only if the specified <search condition> is not *False* for any row of a table.

4.17.4 Domain constraints

A domain constraint is a constraint that is specified for a domain. It is applied to all columns that are based on that domain, and to all values cast to that domain.

A domain constraint is described by a domain constraint descriptor. In addition to the components of every constraint descriptor a domain constraint descriptor includes:

— The <search condition>.

A domain constraint is satisfied by SQL-data if and only if, for any table *T* that has a column named *C* based on that domain, the specified <search condition>, with each occurrence of VALUE replaced by *C*, is not *False* for any row of *T*.

A domain constraint is satisfied by the result of a <cast specification> if and only if the specified <search condition>, with each occurrence of VALUE replaced by that result, is not *False*.

4.17.5 Assertions

An assertion is a named constraint that may relate to the content of individual rows of a table, to the entire contents of a table, or to a state required to exist among a number of tables.

An assertion is described by an assertion descriptor. In addition to the components of every constraint descriptor an assertion descriptor includes:

— The <search condition>.

An assertion is satisfied if and only if the specified <search condition> is not *False*.

4.18 Functional dependencies

4.18.1 Overview of functional dependency rules and notations

This Subclause defines *functional dependency* and specifies a minimal set of rules that a conforming implementation shall follow to determine functional dependencies and candidate keys in base tables and <query expression>s.

The rules in this Subclause may be freely augmented by implementation-defined rules, where indicated in this Subclause.

Let *T* be any table. Let *CT* be the set comprising all the columns of *T*, and let *A* and *B* be arbitrary subsets of *CT*, not necessarily disjoint and possibly empty.

Let "*T*: $A \mapsto B$" (read "in *T*, *A* determines *B*" or "*B* is functionally dependent on *A* in *T*") denote the functional dependency of *B* on *A* in *T*, which is true if, for any possible value of *T*, any two rows that are not distinct for

every column in *A* also are not distinct for every column in *B*. When the table *T* is understood from context, the abbreviation "*A* ↦ *B*" may also be used.

If *X* ↦ *Y* is some functional dependency in some table *T*, then *X* is a *determinant* of *Y* in *T*.

Let *A* ↦ *B* and *C* ↦ *D* be any two functional dependencies in *T*. The following are also functional dependencies in *T*:

— *A* UNION (*C* DIFFERENCE *B*) ↦ *B* UNION *D*

— *C* UNION (*A* DIFFERENCE *D*) ↦ *B* UNION *D*

NOTE 31 — Here, "UNION" denotes set union and "DIFFERENCE" denotes set difference.

These two rules are called the *rules of deduction* for functional dependencies.

Every table has an associated non-empty set of functional dependencies.

The set of functional dependencies is non-empty because *X* ↦ *X* for any *X*. A functional dependency of this form is an axiomatic functional dependency, as is *X* ↦ *Y* where *Y* is a subset of *X*. *X* ↦ *Y* is a non-axiomatic functional dependency if *Y* is not a subset of *X*.

4.18.2 General rules and definitions

In the following Subclauses, let a column *C1* be a *counterpart* of a column *C2* under qualifying table *QT* if *C1* is specified by a column reference (or by a <value expression> that is a column reference) that references *C2* and *QT* is the qualifying table of *C2*. If *C1* is a counterpart of *C2* under qualifying table *QT1* and *C2* is a counterpart of *C3* under qualifying table *QT2*, then *C1* is a counterpart of *C3* under *QT2*.

The notion of counterparts naturally generalizes to sets of columns, as follows: If *S1* and *S2* are sets of columns, and there is a one-to-one correspondence between *S1* and *S2* such that each element of *S1* is a counterpart of the corresponding element of *S2*, then *S1* is a counterpart of *S2*.

The next Subclauses recursively define the notion of *known functional dependency*. This is a ternary relationship between a table and two sets of columns of that table. This relationship expresses that a functional dependency in the table is known to the SQL-implementation. All axiomatic functional dependencies are known functional dependencies. In addition, any functional dependency that can be deduced from known functional dependencies using the rules of deduction for functional dependency is a known functional dependency.

The next Subclauses also recursively define the notion of a "*BUC-set*", which is a set of columns of a table (as in "*S* is BUC-set", where *S* is a set of columns).

NOTE 32 — "BUC" is an acronym for "base table unique constraint", since the starting point of the recursion is a set of known not null columns comprising a nondeferrable unique constraint of a base table.

The notion of BUC-set is closed under the following deduction rule for BUC-sets: If *S1* and *S2* are sets of columns, *S1* is a subset of *S2*, *S1* ↦ *S2*, and *S2* is a BUC-set, then *S1* is also a BUC-set.

NOTE 33 — A BUC-set may be empty, in which case there is at most one row in the table. This case shall be distinguished from a table with no BUC-set.

An SQL-implementation may define additional rules for determining BUC-sets, provided that every BUC-set *S* of columns of a table *T* shall have an associated base table *BT* such that every column of *S* has a counterpart

in *BT*, and for any possible value of the columns of *S*, there is at most one row in *BT* having those values in those columns.

The next Subclauses also recursively define the notion of a "*BPK-set*", which is a set of columns of a table (as in "*S* is a BPK-set", where *S* is a set of columns). Every BPK-set is a BUC-set.

NOTE 34 — "BPK" is an acronym for "base table primary key", since the starting point of the recursion is a set of known not null columns comprising a nondeferrable primary key constraint of a base table.

The notion of BPK-set is closed under the following deduction rule for BPK-sets: If *S1* and *S2* are sets of columns, *S1* is a subset of *S2*, *S1* ↦ *S2*, and *S2* is a BPK-set, then *S1* is also a BPK-set.

NOTE 35 — Like BUC-sets, a BPK-set may be empty.

An SQL-implementation may define additional rules for determining BPK-sets, provided that every BPK-set *S* is a BUC-set, and every member of *S* has a counterpart to a column in a primary key in the associated base table *BT*.

All applicable syntactic transformations (for example, to remove *, CUBE, or ROLLUP) shall be applied before using the rules to determine known functional dependencies, BUC-sets, and BPK-sets.

The following Subclauses use the notion of *AND-component* of a <search condition> *SC*. which is defined recursively as follows:

— If *SC* is a <boolean test> *BT*, then the only AND-component of *SC* is *BT*.

— If *SC* is a <boolean factor> *BF*, then the only AND-component of *SC* is *BF*.

— If *SC* is a <boolean term> of the form "*P* AND *Q*", then the AND-components of *SC* are the AND-components of *P* and the AND-components of *Q*.

— If *SC* is a <boolean value expression> *BVE* that specifies OR, then the only AND-component of *SC* is *BVE*.

Let *AC* be an AND-component of *SC* such that *AC* is a <comparison predicate> whose <comp op> is <equals operator>. Let *RVE1* and *RVE2* be the two <row value predicand>s that are the operands of *AC*. Suppose that both *RVE1* and *RVE2* are <row value constructor predicand>s. Let *n* be the degree of *RVE1*. Let $RVEC1_i$ and $RVEC2_i$, 1 (one) ≤ *i* ≤ *n*, be the *i*-th <common value expression>, <boolean predicand>, or <row value constructor element> of *RVE1* and *RVE2*, respectively. The <comparison predicate> "$RVEC1_i = RVEC2_i$" is called an *equality AND-component* of *SC*.

4.18.3 Known functional dependencies in a base table

Let *T* be a base table and let *CT* be the set comprising all the columns of *T*.

A set of columns *S1* of *T* is a *BPK-set* if it is the set of columns enumerated in some unique constraint *UC* of *T*, *UC* specifies PRIMARY KEY, and *UC* is nondeferrable.

A set of columns *S1* of *T* is a *BUC-set* if it is the set of columns enumerated in some unique constraint *UC* of *T*, *UC* is nondeferrable, and every member of *S1* is known not null.

If *UCL* is a set of columns of *T* such that *UCL* is a BUC-set, then *UCL* ↦ *CT* is a *known functional dependency* in *T*.

If *GC* is a generated column of *T*, then $D \mapsto GC$, where *D* is the set of parameteric columns of *GC*, is a *known functional dependency* in *T*.

Implementation-defined rules may determine other known functional dependencies in *T*.

4.18.4 Known functional dependencies in a transition table

Let *TT* be the transition table defined in a trigger *TR* and let *T* be the subject table of *TR*. If *TT* is an old transition table or if *TR* is an AFTER trigger and *TT* is a new transition table, then the BPK-sets, BUC-sets, and known functional dependencies of *TT* are the same as the BPK-sets, BUC-sets, and known functional dependencies of *T*. If *TR* is a BEFORE trigger and *TT* is a new transition table, then no set of columns of *TT* is a BPK-set or a BUC-set and the known functional dependencies of *TT* are the axiomatic functional dependencies.

4.18.5 Known functional dependencies in <table value constructor>

Let *R* be the result of a <table value constructor>, and let *CR* be the set comprising all the columns of *R*.

No set of columns of *R* is a BPK-set or a BUC-set, except as determined by implementation-defined rules.

All axiomatic functional dependencies are *known functional dependencies* of a <table value constructor>. In addition, there may be implementation-defined known functional dependencies (for example, by examining the actual value of the <table value constructor>).

4.18.6 Known functional dependencies in a <joined table>

Let *T1* and *T2* denote the tables identified by the first and second <table reference>s of some <joined table> *JT*. Let *R* denote the table that is the result of *JT*. Let *CT* be the set of columns of the result of *JT*.

Every column of *R* has some counterpart in either *T1* or *T2*. If NATURAL is specified or the <join specification> is a <named columns join>, then some columns of *R* may have counterparts in both *T1* and *T2*.

A set of columns *S* of *R* is a *BPK-set* if *S* has some counterpart in *T1* or *T2* that is a BPK-set, every member of *S* is known not null, and $S \mapsto CT$ is a *known functional dependency* of *R*.

A set of columns *S* of *R* is a *BUC-set* if *S* has some counterpart in *T1* or *T2* that is a BUC-set, every member of *S* is known not null, and $S \mapsto CT$ is a *known functional dependency* of *R*.

NOTE 36 — The following rules for known functional dependencies in a <joined table> are not mutually exclusive. The set of known functional dependencies is the union of those dependencies generated by all applicable rules, including the rules of deduction presented earlier.

If <join condition> is specified, *AP* is an equality AND-component of the <search condition>, one comparand of *AP* is a column reference *CR*, and the other comparand of *AP* is a <literal>, then let *CRC* be the counterparts of *CR* in *R*. Let { } denote the empty set. $\{\} \mapsto \{CRC\}$ is a *known functional dependency* in *R* if any of the following conditions is true:

— INNER is specified.

— If LEFT is specified and *CR* is a column reference to a column in *T1*.

— If RIGHT is specified and *CR* is a column reference to a column in *T2*.

NOTE 37 — An SQL-implementation may also choose to recognize {} -> {*CRC*} as a known functional dependency if the other comparand is a deterministic expression containing no column references.

If <join condition> is specified, *AP* is an equality AND-component of the <search condition>, one comparand of *AP* is a column reference *CRA*, and the other comparand of *AP* is a column references *CRB*, then let *CRAC* and *CRBC* be the counterparts of *CRA* and *CRB* in *R*. {*CRAC*} ↦ {*CRBC*} is a *known functional dependency* in *R* if any of the following conditions is true:

— INNER is specified.

— If LEFT is specified and *CRA* is a column reference to a column in *T1*.

— If RIGHT is specified and *CRA* is a column reference to a column in *T2*.

NOTE 38 — An SQL-implementation may also choose to recognize the following as known functional dependencies: {*CRAC*} ↦ {*CRBC*} if *CRA* is known not nullable, *CRA* is a column of *T1*, and RIGHT or FULL is specified; or if *CRA* is known not nullable, *CRA* is a column of *T2*, and LEFT or FULL is specified.

NOTE 39 — An SQL-implementation may also choose to recognize similar known functional dependencies of the form { CRA_1 , ..., CRA_N } ↦ {*CRBC*} in case one comparand is a deterministic expression of column references CRA_1 , ..., CRA_N under similar conditions.

If NATURAL is specified, or if a <join specification> immediately containing a <named columns join> is specified, then let C_1 , ..., C_N be the column names of the corresponding join columns, for i between 1 (one) and *N*. Let *SC* be the <search condition>:

```
  ( TN1.C1 = TN2.C1 )
AND
...
AND
( TN1.CN = TN2.CN )
```

Let *SLCC* and *SL* be the <select list>s defined in the Syntax Rules of Subclause 7.7, "<joined table>". Let *JT* be the <join type>. Let *TN1* and *TN2* be the exposed <table or query name> or <correlation name> of tables *T1* and *T2*, respectively. Let *IR* be the result of the <query expression>:

```
SELECT SLCC, TN1.*, TN2.*
FROM TN1 JT JOIN TN2
     ON SC
```

The following are recognized as additional *known functional dependencies* of *IR*:

— If INNER or LEFT is specified, then { COALESCE ($TN1.C_i$, $TN2.C_i$) } ↦ { $TN1.C_i$ }, for all *i* between 1 (one) and *N*.

— If INNER or RIGHT is specified, then { COALESCE ($TN1.C_i$, $TN2.C_i$) } ↦ { $TN2.C_i$ }, for all *i* between 1 (one) and *N*.

The *known functional dependencies* of *R* are the known functional dependencies of:

```
SELECT SL FROM IR
```

4.18.7 Known functional dependencies in a <table primary>

Let *R* be the result of some <table primary> *TP*. The BPK-sets, BUC-sets, and functional dependencies of *R* are determined as follows:

Case:

— If *TP* immediately contains a <table or query name> *TQN* (with or without ONLY), then the counterparts of the BPK-sets and BUC-sets of *TQN* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in the result of *TQN*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

— If *TP* immediately contains a <derived table> *DT*, then the counterparts of the BPK-sets and BUC-sets of *DT* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in the result of *DT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

— If *TP* immediately contains a <lateral derived table> *LDT*, then the counterparts of the BPK-sets and BUC-sets of *LDT* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in the result of *LDT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

— If *TP* immediately contains a <collection derived table> *CDT*, and WITH ORDINALITY is specified, then let *C1* and *C2* be the two columns names of *CDT*. $\{C2\}$ is a BPK-set and a BUC-set, and $\{C2\} \mapsto \{C2, C1\}$ is a *known functional dependency* . If WITH ORDINALITY is not specified, then these rules do not identify any BPK-set, BUC-set, or non-axiomatic known functional dependency.

4.18.8 Known functional dependencies in a <table factor>

Let *R* be the result of <table factor> *TF*. Let *S* be the result of <table primary> immediately contained in *TF*. The counterparts of the BPK-sets and BUC-sets of *S* are the BPK-sets and BUCsets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in *S*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

4.18.9 Known functional dependencies in a <table reference>

Let *R* be the result of some <table reference> *TR*. The BPK-sets, BUC-sets, and functional dependencies of *R* are determined as follows:

Case:

— If *TR* immediately contains a <table factor> *TF*, then the counterparts of the BPK-sets and BUC-sets of *TF* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in the result of *TF*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

— If *TR* immediately contains a <joined table> *JT*, then the counterparts of the BPK-sets and BUC-sets of *JT* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in the result of *JT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

4.18.10 Known functional dependencies in the result of a <from clause>

Let *R* be the result of some <from clause> *FC*.

If there is only one <table reference> *TR* in *FC*, then the counterparts of the BPK-sets of *TR* and the counterparts of the BUC-sets of *TR* are the BPK-sets and BUC-sets of *TR*, respectively. Otherwise, these rules do not identify any BPK-sets or BUC-sets in the result of *FC*.

If *T* is a <table reference> immediately contained in the <table reference list> of *FC*, then all known functional dependencies in *T* are *known functional dependencies* in *R*.

4.18.11 Known functional dependencies in the result of a <where clause>

Let *T* be the table that is the operand of the <where clause>. Let *R* be the result of the <where clause>. A set of columns *S* in *R* is a *BUC-set* if there is a <table reference> *TR* such that every member of *S* has a counterpart in *TR*, the counterpart of *S* in *TR* is a BUC-set, and $S \mapsto CR$, where *CR* is the set of all columns of *R*. If, in addition, the counterpart of *S* is a BPK-set, then *S* is a *BPK-set*.

If $A \mapsto B$ is a known functional dependency in *T*, then let *AC* be the set of columns of *R* whose counterparts are in *A*, and let *BC* be the set of columns of *R* whose counterparts are in *B*. $AC \mapsto BC$ is a *known functional dependency* in *R*.

If *AP* is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of *AP* is a column reference *CR*, and the other comparand of *AP* is a <literal>, then let *CRC* be the counterpart of *CR* in *R*. $\{\} \mapsto \{CRC\}$ is a *known functional dependency* in R, where $\{\}$ denotes the empty set.

NOTE 40 — An SQL-implementation may also choose to recognize $\{\} \mapsto \{CRC\}$ as a known functional dependency if the other comparand is a deterministic expression containing no column references.

If *AP* is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of *AP* is a column reference *CRA*, and the other comparand of *AP* is a column references *CRB*, then let *CRAC* and *CRBC* be the counterparts of *CRA* and *CRB* in *R*. $\{CRBC\} \mapsto \{CRAC\}$ and $\{CRAC\} \mapsto \{CRBC\}$ are *known functional dependencies* in *R*.

NOTE 41 — An SQL-implementation may also choose to recognize known functional dependencies of the form $\{CRAC_1, ..., CRAC_N\} \mapsto \{CRBC\}$ if one comparand is a deterministic expressions that contains column references $CRA_1, ..., CRA_N$ and the other comparand is a column reference *CRB*.

4.18.12 Known functional dependencies in the result of a <group by clause>

Let *T1* be the table that is the operand of the <group by clause>, and let *R* be the result of the <group by clause>.

Let *G* be the set of columns specified by the <grouping column reference list> of the <group by clause>, after applying all syntactic transformations to eliminate ROLLUP, CUBE, and GROUPING SETS.

The columns of *R* are the columns of *G*, with an additional column *CI*, whose value in any particular row of *R* somehow denotes the subset of rows of *T1* that is associated with the combined value of the columns of *G* in that row.

If every element of *G* is a column reference to a known not null column, then *G* is a *BUC-set* of *R*. If *G* is a subset of a BPK-set of columns of *T1*, then *G* is a *BPK-set* of *R*.

$G \mapsto CI$ is a *known functional dependency* in *R*.

NOTE 42 — Any <set function specification> that is specified in conjunction with *R* is necessarily a function of *CI*. If *SFVC* denotes the column containing the results of such a <set function specification>, then $CI \mapsto SFVC$ holds true, and it follows that $G \mapsto SFVC$ is a *known functional dependency* in the table containing *SFVC*.

4.18.13 Known functional dependencies in the result of a <having clause>

Let *T1* be the table that is the operand of the <having clause>, let *SC* be the <search condition> simply contained in the <having clause>, and let *R* be the result of the <having clause>.

If *S* is a set of columns of *R* and the counterpart of *S* in *T1* is a BPK-set, then *S* is a *BPK-set*. If the counterpart of *S* in *T1* is a BUC-set, then *S* is a *BUC-set*.

Any known functional dependency in the <query expression>

SELECT * FROM *T1* WHERE *SC*

is a *known functional dependency* in *R*.

4.18.14 Known functional dependencies in a <query specification>

Let *T* be the <table expression> simply contained in the <query specification> *QS* and let *R* be the result of the <query specification>.

Let *SL* be the <select list> of the <query specification>.

Let *T1* be *T* extended to the right with columns arising from <value expression>s contained in the <select list>, as follows: A <value expression> *VE* that is not a column reference specifies a computed column *CC* in *T1*. For every row in *T1*, the value in *CC* is the result of *VE*.

Let *S* be a set of columns of *R* such that every element of *S* arises from the use of <asterisk> in *SL* or by the specification of a column reference as a <value expression> simply contained in *SL*. *S* has counterparts in *T* and *T1*. If the counterpart of *S* in *T* is a BPK-set, then *S* is a *BPK-set*. If the counterpart of *S* in *T* is a BUC-set or a BPK-set, then *S* is a *BUC-set*.

If $A \mapsto B$ is some known functional dependency in *T*, then $A \mapsto B$ is a *known functional dependency* in *T1*.

Let *CC* be the column specified by some <value expression> *VE* that is not possibly non-deterministic in the <select list>.

Let *OP1*, *OP2*, ... be the operands of *VE* that are column references whose qualifying query is *QS* and that are not contained in an aggregated argument of a <set function specification>.

If *VE* does not contain a <set function specification> whose aggregation query is *QS*, then $\{OP1, OP2, ...\} \mapsto CC$ is a *known functional dependency* in *T1*.

If *VE* contains a <set function specification> whose aggregation query is *QS*, then let $\{G1, ...\}$ be the set of grouping columns of *T*. $\{G1, ..., OP1, OP2, ...\} \mapsto CC$ is a *known functional dependency* in *T1*.

Let $C \mapsto D$ be some known functional dependency in *T1*. If all the columns of *C* have counterparts in *R*, then let *DR* be the set comprising those columns of *D* that have counterparts in *R*. $C \mapsto DR$ is a *known functional dependency* in *R*.

4.18.15 Known functional dependencies in a <query expression>

If a <with clause> is specified, and RECURSIVE is not specified, then the *BPK-sets*, *BUC-sets*, and *known functional dependencies* of the table identified by a <query name> in the <with list> are the same as the BPK-sets, BUC-sets, and known functional dependencies of the corresponding <query expression>, respectively. If RECURSIVE is specified, then the BPK-sets, BUC-sets, and non-axiomatic known functional dependencies are implementation-defined.

A <query expression> that is a <query term> that is a <query primary> that is a <simple table> or a <joined table> is covered by previous Subclauses of this Clause.

If the <query expression> specifies UNION, EXCEPT or INTERSECT, then let *T1* and *T2* be the left and right operand tables and let *R* be the result. Let *CR* be the set comprising all the columns of *R*.

Each column of *R* has a counterpart in *T1* and a counterpart in *T2*.

Case:

— If EXCEPT is specified, then a set *S* of columns of *R* is a *BPK-set* if its counterpart in *T1* is a BPK-set. *S* is a BUC-set if its counterpart in *T1* is a BUC-set.

— If UNION is specified, then there are no BPK-sets and no BUC-sets.

— If INTERSECT is specified, then a set *S* of columns of *R* is a *BPK-set* if either of its counterparts in *T1* and *T2* is a BPK-set. *S* is a *BUC-set* if either of its counterparts in *T1* and *T2* is a BUC-set.

Case:

— If UNION is specified, then no non-axiomatic functional dependency in *T1* or *T2* is a known functional dependency in *R*, apart from any functional dependencies determined by implementation-defined rules.

— If EXCEPT is specified, then all known functional dependencies in *T1* are *known functional dependencies* in *R*.

— If INTERSECT is specified, then all known functional dependencies in *T1* and all known functional dependencies in *T2* are *known functional dependencies* in *R*.

NOTE 43 — Other known functional dependencies may be determined according to implementation-defined rules.

4.19 Candidate keys

If the functional dependency $CK \mapsto CT$ holds true in some table *T*, where *CT* consists of all columns of *T*, and there is no proper subset *CK1* of *CK* such that $CK1 \mapsto CT$ holds true in *T*, then *CK* is a *candidate key* of *T*. The set of candidate keys *SCK* is nonempty because, if no proper subset of *CT* is a candidate key, then *CT* is a candidate key.

NOTE 44 — Because a candidate key is a set (of columns), *SCK* is therefore a set of sets (of columns).

A candidate key *CK* is a *strong candidate key* if *CK* is a BUC-set, or if *T* is a grouped table and *CK* is a subset of the set of grouping columns of *T*. Let *SSCK* be the set of strong candidate keys.

Let *PCK* be the set of *P* such that *P* is a member of *SCK* and *P* is a *BPK-set*.

Case:

— If *PCK* is nonempty, then the *primary key* is chosen from *PCK* as follows: If *PCK* has exactly one element, then that element is the primary key; otherwise, the left-most element of *PCK* is chosen according to the "left-most rule" below. The primary key is also the *preferred candidate key*.

— Otherwise, there is no primary key and the *preferred candidate key* is chosen as follows:

  Case:

  - If *SSCK* has exactly one element, then it is the preferred candidate key; otherwise, if *SSCK* has more than one element, then the left-most element of *SSCK* is chosen, according to the "left-most" rule below.
  - Otherwise, if *SCK* has exactly one element, then it is the preferred candidate key; otherwise, the left-most element of *SCK* is chosen, according to the "left-most" rule below.

— The "left-most" rule:

  - This rule uses the ordering of the columns of a table, as specified elsewhere in this part of ISO/IEC 9075.

    To determine the left-most of two sets of columns of *T*, first list each set in the order of the column-numbers of its members, extending the shorter list with zeros to the length of the longer list. Then, starting at the left of each ordered list, step forward until a pair of unequal column numbers, one from the same position in each list, is found. The list containing the number that is the smaller member of this pair identifies the left-most of the two sets of columns of *T*.

    To determine the left-most of more than two sets of columns of *T*, take the left-most of any two sets, then pair that with one of the remaining sets and take the left-most, and so on until there are no remaining sets.

4.20 SQL-schemas

An SQL-schema is a persistent descriptor that includes:

— The name of the SQL-schema.

— The <authorization identifier> of the owner of the SQL-schema.

— The name of the default character set for the SQL-schema.

— The <schema path specification> defining the SQL-path for SQL-invoked routines for the SQL-schema.

— The descriptor of every component of the SQL-schema.

In this part of ISO/IEC 9075, the term "schema" is used only in the sense of SQL-schema. The persistent objects described by the descriptors are said to be *owned by* or to have been *created by* the <authorization identifier> of the schema. Each component descriptor is one of:

— A domain descriptor.

— A base table descriptor.

— A view descriptor.

— A constraint descriptor.

— A privilege descriptor.

— A character set descriptor.

— A collation descriptor.

— A transliteration descriptor.

— A user-defined type descriptor.

— A routine descriptor.

— A sequence generator descriptor.

A schema is created initially using a <schema definition> and may be subsequently modified incrementally over time by the execution of <SQL schema statement>s. <schema name>s are unique within a catalog.

A <schema name> is explicitly or implicitly qualified by a <catalog name> that identifies a catalog.

Base tables and views are identified by <table name>s. A <table name> consists of a <schema name> and an <identifier>. The <schema name> identifies the schema in which a persistent base table or view identified by the <table name> is defined. Base tables and views defined in different schemas can have <identifier>s that are equal according to the General Rules of Subclause 8.2, "<comparison predicate>".

If a reference to a <table name> does not explicitly contain a <schema name>, then a specific <schema name> is implied. The particular <schema name> associated with such a <table name> depends on the context in which the <table name> appears and is governed by the rules for <schema qualified name>.

If a reference to an SQL-invoked routine that is contained in a <routine invocation> does not explicitly contain a <schema name>, then the SQL-invoked routine is selected from the SQL-path of the schema.

The *containing schema* of an <SQL schema statement> is defined as the schema identified by the <schema name> implicitly or explicitly contained in the name of the object that is created or manipulated by that SQL-statement.

4.21 Sequence generators

4.21.1 General description of sequence generators

A *sequence generator* is a mechanism for generating successive exact numeric values, one at a time. A sequence generator is either an *external sequence generator* or an *internal sequence generator*. An external sequence generator is a named schema object while an internal sequence generator is a component of another schema object. A sequence generator has a data type, which shall be an exact numeric type with scale 0 (zero), a minimum value, a maximum value, a start value, an increment, and a cycle option.

Specification of a sequence generator can optionally include the specification of a data type, a minimum value, a maximum value, a start value, an increment, and a cycle option.

If a sequence generator is associated with a negative increment, then it is a *descending sequence generator*; otherwise, it is an *ascending sequence generator*.

A sequence generator has a time-varying *current base value*, which is a value of its data type. A sequence generator has a cycle which consists of all the possible values between the minimum value and the maximum value which are expressible as (current base value + N * increment), where N is a non-negative number.

When a sequence generator is created, its current base value is initialized to the start value. Subsequently, the current base value is set to the value of the lowest non-issued value in the cycle for an ascending sequence generator, or the highest non-issued value in the cycle for a descending sequence generator.

Any time after a sequence generator is created, its current base value can be set to an arbitrary value of its data type by an <alter sequence generator statement>.

Changes to the current base value of a sequence generator are not controlled by SQL-transactions; therefore, commits and rollbacks of SQL-transactions have no effect on the current base value of a sequence generator.

A sequence generator is described by a sequence generator descriptor. A sequence generator descriptor includes:

— The sequence generator name that is a schema-qualified sequence generator name for an external sequence generator and a zero-length character string for an internal sequence generator.

— The data type descriptor of the data type associated with the sequence generator.

— The increment of the sequence generator.

— The maximum value of the sequence generator.

— The minimum value of the sequence generator.

— The cycle option of the sequence generator.

— The current base value of the sequence generator.

4.21.2 Operations involving sequence generators

When a <next value expression> is applied to a sequence generator *SG*, *SG* issues a value *V* taken from *SG*'s current cycle such that *V* is expressible as the current base value of *SG* plus *N* multiplied by the increment of *SG*, where *N* is a non-negative number.

Thus a sequence generator will normally issue all of the values in its cycle and these will normally be in increasing or decreasing order (depending on the sign of the increment) but within that general ordering separate subgroups of ordered values may occur.

If the sequence generator's cycle is exhausted (*i.e.*, it cannot issue a value that meets the criteria), then a new cycle is created with the current base value set to the minimum value of *SG* (if *SG* is an ascending sequence generator) or the maximum value of *SG* (if *SG* is a descending sequence generator).

If a new cycle is created and the descriptor of *SG* includes NO CYCLE, then an exception condition is raised.

If there are multiple instances of <next value expression>s specifying the same sequence generator within a single SQL-statement, all those instances return the same value for a given row processed by that SQL-statement.

4.22 SQL-client modules

An *SQL-client module* is an SQL-environment object that can include externally-invoked procedures and certain descriptors. An SQL-client module is created and destroyed by implementation-defined mechanisms (which can include the granting and revoking of privileges required for the use of the SQL-client module). An SQL-client module exists in the SQL-environment containing an SQL-client.

If an SQL-client module *S* is defined by an <SQL-client module definition> that contains a <module authorization identifier> *MAI*, then the owner of *S* is *MAI*; otherwise, *S* has no owner.

An SQL-client module can be specified by a <SQL-client module definition> (see Subclause 13.1, "<SQL-client module definition>").

An SQL-client module includes:

— The name, if any of the SQL-client module.

— The name of the standard programming language from a compilation unit of which an externally-invoked procedure included in the module can be invoked.

— The <module authorization identifier>, if any.

— An indication of whether or not the <module authorization identifier> is to apply to execution of prepared statements resulting from invocation of externally-invoked procedures in the SQL-client module that contain <prepare statement>s or <execute immediate statement>s.

— SQL-client module defaults, for use in the application of Syntax Rules to <externally-invoked procedure>s, <temporary table declaration>s, and <declare cursor>s.

- The name of the schema for use as the default <schema name> when deriving externally-invoked procedures from <externally-invoked procedure>s, specified either by the <schema name> or, failing that, by the <module authorization identifier>.

- The SQL-path, if any, used to qualify:
  - — Unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in the SQL-client module.
  - — Unqualified <user-defined type name>s that are immediately contained in <path-resolved user-defined type name>s that are contained in the SQL-client module.
- The names of zero or more *SQL-client module collations*, each specifying a collation for one or more character sets for the SQL-client module.

— The name, if specified, of the character set used to express the <SQL-client module definition>.

NOTE 45 — The <module character set specification> has no effect on the SQL language contained in the SQL-client module and exists only for compatibility with ISO/IEC 9075:1992. It may be used to document the character set of the SQL-client module.

— Module contents:

- Zero or more temporary table descriptors.
- Zero or more cursors.
- One or more externally-invoked procedures.

A compilation unit is a segment of executable code, possibly consisting of one or more subprograms. An SQL-client module is associated with a compilation unit during its execution. A single SQL-client module may be associated with multiple compilation units and multiple SQL-client modules may be associated with a single compilation unit. The manner in which this association is specified, including the possible requirement for execution of some implementation-defined statement, is implementation-defined. Whether a compilation unit may invoke or transfer control to other compilation units, written in the same or a different programming language, is implementation-defined.

4.23 Embedded syntax

An <embedded SQL host program> (<embedded SQL Ada program>, <embedded SQL C program>, <embedded SQL COBOL program>, <embedded SQL Fortran program>, <embedded SQL MUMPS program>, <embedded SQL Pascal program>, or <embedded SQL PL/I program>) is a compilation unit that consists of programming language text and SQL text. The programming language text shall conform to the requirements of a specific standard programming language. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s, as defined in this International Standard. This allows database applications to be expressed in a hybrid form in which SQL-statements are embedded directly in a compilation unit. Such a hybrid compilation unit is defined to be equivalent to:

— An SQL-client module, containing externally-invoked procedures and declarations.

— A standard compilation unit in which each SQL-statement has been replaced by an invocation of an externally-invoked procedure in the SQL-client module, and the declarations contained in such SQL-statements have been suitably transformed into declarations in the host language.

If an <embedded SQL host program> contains an <embedded authorization declaration>, then it shall be the first statement or declaration in the <embedded SQL host program>. The <embedded authorization declaration> is not replaced by a procedure or subroutine call of an <externally-invoked procedure>, but

is removed and replaced by syntax associated with the <SQL-client module definition>'s <module authorization clause>.

An implementation may reserve a portion of the name space in the <embedded SQL host program> for the names of procedures or subroutines that are generated to replace SQL-statements and for program variables and branch labels that may be generated as required to support the calling of these procedures or subroutines; whether this reservation is made is implementation-defined. They may similarly reserve name space for the <SQL-client module name> and <procedure name>s of the generated <SQL-client module definition> that may be associated with the resulting standard compilation unit. The portion of the name space to be so reserved, if any, is implementation-defined.

4.24 Dynamic SQL concepts

4.24.1 Overview of dynamic SQL

In many cases, the SQL-statement to be executed can be coded into an <SQL-client module definition> or into a compilation unit using the embedded syntax. In other cases, the SQL-statement is not known when the program is written and will be generated during program execution.

Dynamic execution of SQL-statements can generally be accomplished in two different ways. Statements can be *prepared* for execution and then later executed one or more times; when the statement is no longer needed for execution, it can be *released* by the use of a <deallocate prepared statement>. Alternatively, a statement that is needed only once can be executed without the preparation step—it can be *executed immediately* (not all SQL-statements can be executed immediately).

When a prepared statement is executed, if it has an owner, then it is executed under definer's rights; otherwise, it is executed under invoker's rights.

Many SQL-statements can be written to use "parameters" (which are manifested in static execution of SQL-statements as host parameters in <SQL procedure statement>s contained in <externally-invoked procedure>s in <SQL-client module definition>s or as host variables in <embedded SQL statement>s contained in <embedded SQL host program>s). In SQL-statements that are executed dynamically, the parameters are called dynamic parameters (<dynamic parameter specification>s) and are represented in SQL language by a <question mark> (?).

In many situations, an application that generates an SQL-statement for dynamic execution knows in detail the required characteristics (*e.g.*, <data type>, <length>, <precision>, <scale>, *etc.*) of each of the dynamic parameters used in the statement; similarly, the application may also know in detail the characteristics of the values that will be returned by execution of the statement. However, in other cases, the application may not know this information to the required level of detail; it is possible in some cases for the application to ascertain the information from the Information Schema, but in other cases (*e.g.*, when a returned value is derived from a computation instead of simply from a column in a table, or when dynamic parameters are supplied) this information is not generally available except in the context of preparing the statement for execution.

NOTE 46 — The Information Schema is defined in ISO/IEC 9075-11.

To provide the necessary information to applications, SQL permits an application to request the SQL-server to *describe* a prepared statement. The description of a statement identifies the number of input dynamic

parameters (*describe input*) and their data type information or it identifies the number of output dynamic parameters or values to be returned (*describe output*) and their data type information. The description of a statement is placed into the SQL descriptor areas already mentioned.

Many, but not all, SQL-statements can be prepared and executed dynamically.

NOTE 47 — The complete list of statements that may be dynamically prepared and executed is defined in Subclause 4.33.7, "Preparable and immediately executable SQL-statements".

Certain "set statements" (<set catalog statement>, <set schema statement>, <set names statement>, and <set path statement>) have no effect other than to set up default information (catalog name, schema name, character set, and SQL path, respectively) to be applied to other SQL-statements that are prepared or executed immediately or that are invoked directly.

Syntax errors and Access Rule violations caused by the preparation or immediate execution of <preparable statement>s are identified when the statement is prepared (by <prepare statement>) or when it is executed (by <execute statement> or <execute immediate statement>); such violations are indicated by the raising of an exception condition.

4.24.2 Dynamic SQL statements and descriptor areas

An <execute immediate statement> can be used for a one-time preparation and execution of an SQL-statement. A <prepare statement> is used to prepare the generated SQL-statement for subsequent execution. A <deallocate prepared statement> is used to deallocate SQL-statements that have been prepared with a <prepare statement>. A description of the input dynamic parameters for a prepared statement can be obtained by execution of a <describe input statement>. A description of the resultant columns of a <dynamic select statement> or <dynamic single row select statement> can be obtained by execution of a <describe output statement>. A description of the output dynamic parameters of a statement that is neither a <dynamic select statement> nor a <dynamic single row select statement> can be obtained by execution of a <describe output statement>.

For a statement other than a <dynamic select statement>, an <execute statement> is used to associate parameters with the prepared statement and execute it as though it had been coded when the program was written. For a <dynamic select statement>, the prepared <cursor specification> is associated with a cursor via a <dynamic declare cursor> or <allocate cursor statement>. The cursor can be opened and dynamic parameters can be associated with the cursor with a <dynamic open statement>. A <dynamic fetch statement> positions an open cursor on a specified row and retrieves the values of the columns of that row. A <dynamic close statement> closes a cursor that was opened with a <dynamic open statement>. A <dynamic delete statement: positioned> is used to delete rows through a dynamic cursor. A <dynamic update statement: positioned> is used to update rows through a dynamic cursor. A <preparable dynamic delete statement: positioned> is used to delete rows through a dynamic cursor when the precise format of the statement isn't known until runtime. A <preparable dynamic update statement: positioned> is used to update rows through a dynamic cursor when the precise format of the statement isn't known until runtime.

The interface for input dynamic parameters and output dynamic parameters for a prepared statement and for the resulting values from a <dynamic fetch statement> or the execution of a prepared <dynamic single row select statement> can be either a list of dynamic parameters or embedded variables or an SQL descriptor area. An SQL descriptor area consists of one or more item descriptor areas, together with a header that includes a count of the number of those item descriptor areas. The header of an SQL descriptor area consists of the fields in Table 23, "Data types of <key word>s used in the header of SQL descriptor areas", in Subclause 19.1, "Description of SQL descriptor areas". Each item descriptor area consists of the fields specified in Table 24,

"Data types of <key word>s used in SQL item descriptor areas", in Subclause 19.1, "Description of SQL descriptor areas". The SQL descriptor area is allocated and maintained by the system with the following statements: <allocate descriptor statement>, <deallocate descriptor statement>, <set descriptor statement>, and <get descriptor statement>.

Two kinds of identifier are used for referencing dynamic SQL objects, *extended names* and *non-extended names*. An extended name is an <identifier> assigned to a parameter or variable and the object it identifies is referenced indirectly, by referencing that parameter or variable. A non-extended name is just an <identifier> and the object it identifies is referenced by using that <identifier> directly in an SQL-statement.

SQL descriptor areas are always identified by extended names. Dynamic statements and cursors can be identified either by non-extended names or by extended names.

Two extended names are equivalent if their values, with leading and trailing <space>s removed, are equivalent according to the rules for <identifier> comparison in Subclause 5.2, "<token> and <separator>".

The *scope* of an extended name is either *global* or *local* and is determined by the run-time context in which the object it identifies is brought into existence.

The scope of a global extended name *GEN* is the SQL-session, meaning that, during the existence of the object *O* it identifies, *GEN* can be used to reference *O* by any SQL-statement executed in that SQL-session.

The scope of a local extended name *LEN* is the SQL-client module *M* containing the externally-invoked procedure that is being executed when the object *O* identified by *LEN* is brought into existence. This means that, during the existence of *O*, *LEN* can be used to reference *O* by any SQL-statement executed in the same SQL-session by an externallyinvoked procedure in *M*.

The scope of a non-extended name is the <SQL-client module definition> containing the SQL-statement that defines it.

NOTE 48 — The namespace of non-extended names is different from the namespace of extended names.

Let *PRP* be the prepared statement resulting from execution of a <prepare statement> in an externally-invoked procedure, SQL-invoked routine, or triggered action *E*. In the following cases, *PRP* has no owner:

— *E* is an SQL-invoked routine whose security characteristic is INVOKED.

— *E* is an externally-invoked procedure contained in an SQL-client module that either has no owner or for which FOR STATIC ONLY was specified.

Otherwise, the owner of *PRP* is the owner of *E*.

4.25 Direct invocation of SQL

Direct invocation of SQL is a mechanism for executing direct SQL-statements, known as <direct SQL statement>s. In direct invocation of SQL, the method of invoking <direct SQL statement>s, the method of raising conditions that result from the execution of <direct SQL statement>s, the method of accessing the diagnostics information that results from the execution of <direct SQL statement>s, and the method of returning the results are implementation-defined.

4.26 Externally-invoked procedures

An externally-invoked procedure consists of an SQL-statement and can be invoked from a compilation unit of a host language. The host language is specified by the <language clause> of the SQL-client module that contains the externally-invoked procedure.

4.27 SQL-invoked routines

4.27.1 Overview of SQL-invoked routines

An *SQL-invoked routine* is an SQL-invoked procedure or an SQL-invoked function. An SQL-invoked routine comprises at least a <schema qualified routine name>, a sequence of <SQL parameter declaration>s, and a <routine body>.

An SQL-invoked routine is an element of an SQL-schema and is called a *schema-level routine.*

An SQL-invoked routine *SR* is said to be *dependent* on a user-defined type *UDT* if *SR* is created during the execution of the <user-defined type definition> that created *UDT* or if *SR* is created during the execution of an <alter type statement> that specifies an <add attribute definition>. An SQL-invoked routine that is dependent on a user-defined type cannot be modified by an <alter routine statement> or be destroyed by a <drop routine statement>. It is destroyed implicitly by a <drop data type statement>.

An *SQL-invoked procedure* is an SQL-invoked routine that is invoked from an SQL <call statement>. An SQL-invoked procedure may have input SQL parameters, output SQL parameters, and SQL parameters that are both input SQL parameters and output SQL parameters. The format of an SQL-invoked procedure is specified by <SQL-invoked procedure> (see Subclause 11.50, "<SQL-invoked routine>").

An SQL-invoked procedure may optionally be specified to require a new savepoint level to be established when it is invoked and destroyed on return from the executed routine body. The alternative of not taking a savepoint can also be directly specified with OLD SAVEPOINT LEVEL. When an SQL-invoked function is invoked a new savepoint level is always established. Savepoint levels are described in Subclause 4.35.2, "Savepoints".

An *SQL-invoked function* is an SQL-invoked routine whose invocation returns a value. Every parameter of an SQL-invoked function is an input SQL parameter, one of which may be designated as the result SQL parameter. The format of an SQL-invoked function is specified by <SQL-invoked function> (see Subclause 11.50, "<SQL-invoked routine>"). An SQL-invoked function can be a *type-preserving function*; a type-preserving function is an SQL-invoked function that has a result SQL parameter. The most specific type of a non-null result of invoking a type-preserving function shall be compatible with the most specific type of the value of the argument substituted for its result SQL parameter.

An *SQL-invoked method* is an SQL-invoked function that is specified by <method specification designator> (see Subclause 11.50, "<SQL-invoked routine>"). There are three kinds of SQL-invoked methods: *SQL-invoked constructor methods*, *instance SQL-invoked methods* and *static SQL-invoked methods*. All SQL-invoked methods are associated with a user-defined type, also known as the *type of the method*. The <method characteristic>s of an SQL-invoked method are specified by a <method specification> contained in the <user-defined

type definition> of the type of the method. Both an instance SQL-invoked method and an SQL-invoked constructor method satisfy the following conditions:

— Its first parameter, called the *subject parameter*, has a declared type that is a user-defined type. The type of the subject parameter is the type of the method. A parameter other than the subject parameter is called an *additional parameter*.

— Its descriptor is in the same schema as the descriptor of the data type of its subject parameter.

An SQL-invoked constructor method satisfies the following additional conditions:

— Its <method name> is equivalent to the <qualified identifier> simply contained in the <user-defined type name> included in the user-defined type descriptor of the type of the method.

A static SQL-invoked method satisfies the following conditions:

— It has no subject parameter. Its first parameter, if any, is treated no differently than any other parameter.

— Its descriptor is in the same schema as the descriptor of the structured type of the method. The name of this type (or of some subtype of it) is always specified together with the name of the method when the method is to be invoked.

An SQL-invoked function that is not an SQL-invoked method is an *SQL-invoked regular function*. An SQL-invoked regular function is specified by <function specification> (see Subclause 11.50, "<SQL-invoked routine>").

A *null-call function* is an SQL-invoked function that is defined to return the null value if any of its input arguments is the null value. A null-call function is an SQL-invoked function whose <null-call clause> specifies "RETURNS NULL ON NULL INPUT".

4.27.2 Characteristics of SQL-invoked routines

An SQL-invoked routine can be an *SQL routine* or an *external routine*. An SQL routine is an SQL-invoked routine whose <language clause> specifies SQL. The <routine body> of an SQL routine is an <SQL procedure statement>; the <SQL procedure statement> forming the <routine body> can be any SQL-statement, including an <SQL control statement>, but excluding an <SQL connection statement> and an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>.

An external routine is one whose <language clause> does not specify SQL. The <routine body> of an external routine is an <external body reference> whose <external routine name> identifies a program written in some standard programming language other than SQL. The program identified by <external routine name> shall not execute either an <SQL connection statement> or an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>

An SQL-invoked routine is uniquely identified by a <specific name>, called the *specific name* of the SQL-invoked routine.

SQL-invoked routines are invoked differently depending on their form. SQL-invoked procedures are invoked by <call statement>s. SQL-invoked regular functions are invoked by <routine invocation>s. Instance SQL-invoked methods are invoked by <method invocation>s, while SQL-invoked constructor methods are invoked

by <new specification>s and static SQL-invoked methods are invoked by <static method invocation>s. An invocation of an SQL-invoked routine specifies the <routine name> of the SQL-invoked routine and supplies a sequence of argument values corresponding to the <SQL parameter declaration>s of the SQL-invoked routine. A *subject routine* of an invocation is an SQL-invoked routine that may be invoked by a <routine invocation>. After the selection of the subject routine of a <routine invocation>, the SQL arguments are evaluated and the SQL-invoked routine that will be executed is selected. If the subject routine is an instance SQL-invoked method, then the SQL-invoked routine that is executed is selected from the set of overriding methods of the subject routine. (The term "set of overriding methods" is defined in the General Rules of Subclause 10.4, "<routine invocation>".) The overriding method that is selected is the overriding method with a subject parameter the type designator of whose declared type precedes that of the declared type of the subject parameter of every other overriding method in the type precedence list of the most specific type of the value of the SQL argument that corresponds to the subject parameter. See the General Rules of Subclause 10.4, "<routine invocation>". If the subject routine is not an SQL-invoked method, then the SQL-invoked routine executed is that subject routine. After the selection of the SQL-invoked routine for execution, the values of the SQL arguments are assigned to the corresponding SQL parameters of the SQL-invoked routine and its <routine body> is executed. If the SQL-invoked routine is an SQL routine, then the <routine body> is an <SQL procedure statement> that is executed according to the General Rules of <SQL procedure statement>. If the SQL-invoked routine is an external routine, then the <routine body> identifies a program written in some standard programming language other than SQL that is executed according to the rules of that standard programming language.

The <routine body> of an SQL-invoked routine is always executed under the same SQL-session from which the SQL-invoked routine was invoked. Before the execution of the <routine body>, a new context for the current SQL-session is created and the values of the current context preserved. When the execution of the <routine body> completes the original context of the current SQL-session is restored.

If the SQL-invoked routine is an external routine, then an effective SQL parameter list is constructed before the execution of the <routine body>. The effective SQL parameter list has different entries depending on the parameter passing style of the SQL-invoked routine. The value of each entry in the effective SQL parameter list is set according to the General Rules of Subclause 10.4, "<routine invocation>", and passed to the program identified by the <routine body> according to the rules of Subclause 13.6, "Data type correspondences". After the execution of that program, if the parameter passing style of the SQL-invoked routine is SQL, then the SQL-implementation obtains the values for output parameters (if any), the value (if any) returned from the program, the value of the SQLSTATE, and the value of the message text (if any) from the values assigned by the program to the effective SQL parameter list. If the parameter passing style of the SQL-invoked routine is GENERAL, then such values are obtained in an implementation-defined manner.

Different SQL-invoked routines can have equivalent <routine name>s. No two SQL-invoked functions in the same schema are allowed to have the same signature. No two SQL-invoked procedures in the same schema are allowed to have the same name and the same number of parameters. Subject routine determination is the process for choosing the subject routine for a given <routine invocation> given a <routine name> and an <SQL argument list>. Subject routine determination for SQL-invoked functions considers the most specific types of all of the arguments (that is, all of the arguments that are not <dynamic parameter specification>s whose types are not known at the time of subject routine determination) to the invocation of the SQL-invoked function in order from left to right. Where there is not an exact match between the most specific types of the arguments and the declared types of the parameters, type precedence lists are used to determine the closest match. See Subclause 9.4, "Subject routine determination".

If a <routine invocation> is contained in a <query expression> of a view, a check constraint, or an assertion, the <triggered action> of a trigger, or in an <SQL-invoked routine>, then the subject routine for that invocation is determined at the time the view is created, the check constraint is defined, the assertion is created, the trigger is created, or the SQL-invoked routine is created. If the subject routine is an SQL-invoked procedure, an SQL-

invoked regular function, or a static SQL-invoked method, then the same SQL-invoked routine is executed whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked. If the subject routine is an instance SQL-invoked method, then the SQL-invoked routine that is executed is determined whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked, based on the most specific type of the value resulting from the evaluation of the SQL argument that correspond to the subject parameter. See the General Rules of Subclause 10.4, "<routine invocation>".

All <identifier chain>s in the <routine body> of an SQL routine are resolved to identify the basis and basis referent at the time that the SQL routine is created. Thus, the same columns and SQL parameters are referenced whenever the SQL routine is invoked.

An SQL-invoked routine is either *deterministic* or *possibly non-deterministic*. An SQL-invoked function that is deterministic always returns the identical return value for a given list of SQL argument values. An SQL-invoked procedure that is deterministic always returns the identical values in its output and inout SQL parameters for a given list of SQL argument values. An SQL-invoked routine is possibly non-deterministic if it might produce nonidentical results when invoked with the identical list of SQL argument values.

An external routine *does not possibly contain SQL*, *possibly contains SQL*, *possibly reads SQL-data*, or *possibly modifies SQL-data*. Only an external routine that possibly contains SQL, *possibly reads SQL-data*, or *possibly modifies SQL-data* is permitted to execute SQL-statements during its invocation. Only an SQL-invoked routine that possibly reads SQL-data or possibly modifies SQL-data may read SQL-data during its invocation. Only an SQL-invoked routine that possibly modifies SQL-data may modify SQL-data during its invocation.

An SQL-invoked routine has a *routine authorization identifier*, which is (directly or indirectly) the authorization identifier of the owner of the schema that contains the SQL-invoked routine at the time that the SQL-invoked routine is created.

4.27.3 Execution of SQL-invoked routines

When the <routine body> of an SQL-invoked routine is executed and the new SQL-session context for the SQL-session is created, the SQL-session user identifier in the new SQL-session context is set to the current user identifier in the SQL-session context that was active when the SQL-session caused the execution of the <routine body>. The authorization stack of this new SQL-session context is initially set to empty and a new pair of identifiers is immediately appended to the authorization stack such that:

— The user identifier is the newly initialized SQL-session user identifier.

— The role name is the current role name of the SQL-session context that was active when the SQL-session caused the execution of the <routine body>.

The identifiers in this new entry of the authorization stack are then modified depending on whether the SQL-invoked routine is an SQL routine or an external routine.

If the SQL-invoked routine is an SQL routine, then the identifiers are determined according to the SQL security characteristic of the SQL-invoked routine:

— If the SQL security characteristic is DEFINER, then:

  • If the routine authorization identifier is a user identifier, the user identifier is set to the routine authorization identifier and the role name is set to null.

- Otherwise, the role name is set to the routine authorization identifier and the user identifier is set to null.

— If the SQL security characteristic is INVOKER, then the identifiers remain unchanged.

If the SQL-invoked routine is an external routine, then the identifiers are determined according to the external security characteristic of the SQL-invoked routine:

— If the external security characteristic is DEFINER, then:

- If the routine authorization identifier is a user identifier, then the user identifier is set to the routine authorization identifier and the role name is set to the null value.
- Otherwise, the role name is set to the routine authorization identifier and the user identifier is set to the null value.

— If the external security characteristic is INVOKER, then the identifiers remain unchanged.

— If the external security characteristic is IMPLEMENTATION DEFINED, then the identifiers are set to implementation-defined values.

An SQL-invoked routine that is an external routine also has an *external routine authorization identifier*, which is the <module authorization identifier>, if any, of the <SQL-client module definition> contained in the external program identified by the <routine body> of the external routine. If that <SQL-client module definition> does not specify a <module authorization identifier>, then the external routine authorization identifier is an implementation-defined authorization identifier.

The final value of the user identifier and role name in the authorization stack are used for privilege determination for access to the SQL objects, if any, referenced in the <SQL procedure statement>s that are executed during the execution of the <routine body>.

An SQL-invoked routine has a *routine SQL-path*, which is inherited from its containing SQL-schema, the current SQL-session, or the containing SQL-client module.

An SQL-invoked routine that is an external routine also has an *external routine SQL-path*, which is derived from the <module path specification>, if any, of the <SQL-client module definition> contained in the external program identified by the routine body of the external routine. If that <SQL-client module definition> does not specify a <module path specification>, then the external routine SQL-path is an implementation-defined SQL-path. For both SQL and external routines, the SQL-path of the current SQL-session is used to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>. SQL routines use the routine SQL-path to determine the search order for the subject routines of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>. External routines use the external routine SQL-path to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>.

4.27.4 Routine descriptors

An SQL-invoked routine is described by a *routine descriptor*. A routine descriptor includes:

— The routine name of the SQL-invoked routine.

— The specific name of the SQL-invoked routine.

— The routine authorization identifier of the SQL-invoked routine.

— The routine SQL-path of the SQL-invoked routine.

— The name of the language in which the body of the SQL-invoked routine is written.

— For each of the SQL-invoked routine's SQL parameters, the <SQL parameter name>, if it is specified, the <data type>, the ordinal position, and an indication of whether the SQL parameter is an input SQL parameter, an output SQL parameter, or both an input SQL parameter and an output SQL parameter.

— An indication of whether the SQL-invoked routine is an SQL-invoked function or an SQL-invoked procedure.

— If the SQL-invoked routine is an SQL-invoked procedure, then the maximum number of dynamic result sets.

— An indication of whether the SQL-invoked routine is deterministic or possibly non-deterministic.

— Indications of whether the SQL-invoked routine possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

— If the SQL-invoked routine is an SQL-invoked function, then:

- The <returns data type> of the SQL-invoked function.
- If the <returns data type> simply contains <locator indication>, then an indication that the return value is a locator.
- An indication of whether the SQL-invoked function is a type-preserving function or not.
- An indication of whether the SQL-invoked function is a mutator function or not.
- If the SQL-invoked function is a type-preserving function, then an indication of which parameter is the result parameter.
- An indication of whether the SQL-invoked function is a null-call function.
- An indication of whether the SQL-invoked function is an SQL-invoked method.

— The creation timestamp.

— The last-altered timestamp.

— If the SQL-invoked routine is an SQL routine, then:

- The SQL routine body of the SQL-invoked routine.
- The SQL security characteristic of the SQL routine.

— If the SQL-invoked routine is an external routine, then:

- The external routine name of the external routine.
- The <parameter style> of the external routine.
- If the external routine specifies a <result cast>, then an indication that it specifies a <result cast> and the <data type> specified in the <result cast>. If <result cast> contains <locator indication>, then an indication that the <data type> specified in the <result cast> has a locator indication.
- The external security characteristic of the external routine.
- The external routine authorization identifier of the external routine.
- The external routine SQL-path of the external routine.
- The effective SQL parameter list of the external routine.
- For every SQL parameter that has an associated from-sql function *FSF*, the specific name of *FSF*.
- For every SQL parameter that has an associated to-sql function *TSF*, the specific name of *TSF*.
- If the SQL-invoked routine is an external function and if it has a to-sql function *TRF* associated with the result, then the specific name of *TRF*.
- For every SQL parameter whose <SQL parameter declaration> contains <locator indication>, an indication that the SQL parameter is a locator parameter.

— The schema name of the schema that includes the SQL-invoked routine.

— If the SQL-invoked routine is an SQL-invoked method, then:

- An indication of the user-defined type whose descriptor contains the corresponding method specification descriptor.
- An indication of whether STATIC was specified.

— An indication of whether the SQL-invoked routine is dependent on a user-defined type.

— An indication as to whether or not the SQL-invoked routine requires a new savepoint level to be established when it is invoked.

4.28 SQL-paths

An SQL-path is a list of one or more <schema name>s that determines the search order for one of the following:

— The subject routine of a <routine invocation> whose <routine name> does not contain a <schema name>.

— The user-defined type when the <path-resolved user-defined type name> does not contain a <schema name>.

The value specified by CURRENT_PATH is the value of the SQL-path of the current SQL-session. This SQL-path is used to search for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> when the <routine invocation> is contained in <preparable statement>s that are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement>, or contained

in <direct SQL statement>s that are invoked directly. The definition of SQL-schemas specifies an SQL-path that is used to search for the subject routine of a <routine invocation> whose <routine name>s do not contain a <schema name> when the <routine invocation> is contained in the <schema definition>.

4.29 Host parameters

4.29.1 Overview of host parameters

A host parameter is declared in an <externally-invoked procedure> by a <host parameter declaration>. A host parameter either assumes or supplies the value of the corresponding argument in the invocation of the <externally-invoked procedure>.

A <host parameter declaration> specifies the <data type> of its value, which maps to the host language type of its corresponding argument. Host parameters cannot be null, except through the use of indicator parameters.

4.29.2 Status parameters

The SQLSTATE host parameter is a status parameter. It is set to status codes that indicate either that a call of the <externally-invoked procedure> completed successfully or that an exception condition was raised during execution of the <externally-invoked procedure>.

An <externally-invoked procedure> shall specify the SQLSTATE host parameter. The SQLSTATE host parameter is a character string host parameter for which exception values are defined in Clause 23, "Status codes".

If a condition is raised that causes a statement to have no effect other than that associated with raising the condition (that is, not a completion condition), then the condition is said to be an *exception condition* or *exception*. If a condition is raised that permits a statement to have an effect other than that associated with raising the condition (corresponding to an SQLSTATE class value of *successful completion*, *warning*, or *no data*), then the condition is said to be a *completion condition.*

Exception conditions or completion conditions may be raised during the execution of an <SQL procedure statement>. One of the conditions becomes the active condition when the <SQL procedure statement> terminates. If the active condition is an exception condition, then it will be called the active exception condition. If the active condition is a completion condition, then it will be called the active completion condition.

The completion condition *warning* is broadly defined as completion in which the effects are correct, but there is reason to caution the user about those effects. It is raised for implementation-defined conditions as well as conditions specified in this part of ISO/IEC 9075. The completion condition *no data* has special significance and is used to indicate an empty result. The completion condition *successful completion* is defined to indicate a completion condition that does not correspond to *warning* or *no data*. This includes conditions in which the SQLSTATE subclass provides implementation-defined information of a non-cautionary nature.

For the purpose of choosing status parameter values to be returned, exception conditions for transaction rollback have precedence over exception conditions for statement failure. Similarly, the completion condition *no data*

has precedence over the completion condition *warning*, which has precedence over the completion condition *successful completion*. All exception conditions have precedence over all completion conditions. The values assigned to SQLSTATE shall obey these precedence requirements.

4.29.3 Data parameters

A data parameter is a host parameter that is used to either assume or supply the value of data exchanged between a host program and an SQL-implementation.

4.29.4 Indicator parameters

An indicator parameter is an integer host parameter that is specified immediately following another host parameter. Its primary use is to indicate whether the value that the other host parameter assumes or supplies is a null value. An indicator host parameter cannot immediately follow another indicator host parameter.

The other use for indicator parameters is to indicate whether string data truncation occurred during a transfer between a host program and an SQL-implementation in host parameters or host variables. If a non-null string value is transferred and the length of the target is sufficient to accept the entire source value, then the indicator parameter or variable is set to 0 (zero) to indicate that truncation did not occur. However, if the length of the target is insufficient, the indicator parameter or variable is set to the length (in characters or octets, as appropriate) of the source value to indicate that truncation occurred and to indicate original length in characters or octets, as appropriate, of the source.

4.29.5 Locators

A host parameter, a host variable, an SQL parameter of an external routine, or the value returned by an external function may be specified to be a *locator* by specifying AS LOCATOR. A locator is an SQL-session object, rather than SQL-data, that can be used to reference an SQL-data instance. A locator is either a large object locator, a user-defined type locator, an array locator, or a multiset locator.

A large object locator is one of the following:

— Binary large object locator, a value of which identifies a binary large object.

— Character large object locator, a value of which identifies a large object character string.

— National character large object locator, a value of which identifies a national large object character string.

A user-defined type locator identifies a value of the user-defined type specified by the locator specification. An array locator identifies a value of the array type specified by the locator specification. A multiset locator identifies a value of the multiset type specified by the locator specification.

When the value at a site of binary large object type, character large object type, user-defined type, array type, or multiset type is to be assigned to locator of the corresponding type, an implementation-dependent four-octet

non-zero integer value is generated and assigned to the target. A locator value uniquely identifies a value of the corresponding type.

A locator may be either *valid* or *invalid*. A host parameter or host variable specified as a locator may be further specified to be a *holdable locator*. When a locator is initially created, it is marked valid and, if applicable, not holdable. A <hold locator statement> identifying the locator shall be specifically executed before the end of the SQL-transaction in which it was created in order to make that locator holdable.

A non-holdable locator remains valid until the end of the SQL-transaction in which it was generated, unless it is explicitly made invalid by the execution of a <free locator statement> or a <rollback statement> that specifies a <savepoint clause> is executed before the end of that SQL-transaction if the locator was generated subsequent to the establishment of the savepoint identified by the <savepoint clause>.

A holdable locator may remain valid beyond the end of the SQL-transaction in which it is generated. A holdable locator becomes invalid whenever a <free locator statement> identifying that locator is executed or the SQL-transaction in which it is generated or any subsequent SQL-transaction is rolled back. All locators remaining valid at the end of an SQL-session are marked invalid when that SQL-session terminates.

4.30 Diagnostics area

A diagnostics area is a place where completion and exception condition information is stored when an SQL-statement is executed. The diagnostics areas associated with an SQL-session form the *diagnostics area stack* of that SQL-session. For definitional purposes, the diagnostics areas in this stack are considered to be numbered sequentially beginning with 1 (one). An additional diagnostics area is maintained by the SQL-client, as described in ISO/IEC 9075-1, Subclause 4.2.3.1, "SQL-clients".

Two operations on diagnostics area stacks are specified in this International Standard for definitional purposes only. *Pushing* a diagnostics area stack effectively creates a new first diagnostics area, incrementing the ordinal position of every existing diagnostics area in the stack by 1 (one). The content of the new first diagnostics area is initially a copy of the content of the old (now second) one. *Popping* a diagnostics area stack effectively destroys the first diagnostics area in the stack and decrements the ordinal position of every remaining diagnostics area by 1 (one). The maximum number of diagnostics areas in a diagnostics area stack is implementation-dependent.

Each diagnostics area consists of a *statement area* and a sequence of one or more *condition areas*, each of which is at any particular time either *occupied* or *vacant*. A diagnostics area is *empty* when each of its condition areas is vacant; *emptying* a diagnostics area brings about this state. A statement area consists of a collection of named *statement information items*. A condition area consists of a collection of named *condition information items*.

A statement information item gives information about the innermost SQL-statement that is being executed when a condition is raised. A condition information item gives information about the condition itself. The names and data types of statement and condition information items are given in Table 30, "<identifier>s for use with <get diagnostics statement>". Their meanings are given by the General Rules of Subclause 22.1, "<get diagnostics statement>".

At the beginning of the execution of any <SQL procedure statement> that is not an <SQL diagnostics statement>, the first diagnostics area is emptied. An implementation places information about a completion condition or an exception condition reported by SQLSTATE into a vacant condition area in this diagnostics area. If other

conditions are raised, the extent to which these cause further condition areas to become occupied is implementation-defined.

An <externally-invoked procedure> containing an <SQL diagnostics statement> returns a code indicating a completion or an exception condition for that statement via SQLSTATE, but does not necessarily cause any vacant condition areas to become occupied.

The number of condition areas per diagnostics area is referred to as the *condition area limit*. An SQL-agent may set the condition area limit with the <set transaction statement>; if the SQL-agent does not specify the condition area limit, then the condition area limit is implementation-dependent, but shall be at least one condition area. An SQL-implementation may place information into this area about fewer conditions than there are condition areas. The ordering of the information about conditions placed into a diagnostics area is implementation-dependent, except that the first condition area in a diagnostics area always corresponds to the condition specified by the SQLSTATE value.

The <get diagnostics statement> is used to obtain information from an occupied condition area, referenced by its ordinal position within the first diagnostics area.

4.31 Standard programming languages

This part of ISO/IEC 9075 specifies the actions of <externally-invoked procedure>s in SQL-client modules when those <externally-invoked procedure>s are called by programs that conform to certain specified programming language standards. The term "standard *PLN* program", where *PLN* is the name of a programming language, refers to a program that conforms to the standard for that programming language as specified in Clause 2, "Normative references".

This part of ISO/IEC 9075 specifies a mechanism whereby SQL language may be embedded in programs that otherwise conform to any of the same specified programming language standards.

NOTE 49 — Interfaces between SQL and the Java programming language are defined in ISO/IEC 9075-10 and ISO/IEC 9075-13.

Although there are obviouis mappings between many SQL data types and the data types of most standard programming languages, this is not the case for all SQL data types or for all standard programming languages.

For the purposes of interfacing with programming languages, the data types DATE, TIME, TIMESTAMP, and INTERVAL shall be converted to or from character strings in those programming languages by means of a <cast specification>. It is anticipated that future evolution of programming language standards will support data types corresponding to these four SQL data types; this standard will then be amended to reflect the availability of those corresponding data types.

The data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are also mapped to character strings in the programming languages. However, because the facilities available in the programming languages do not provide the same capabilities as those available in SQL, there shall be agreement between the host program and SQL regarding the specific format of the character data being exchanged. Specific syntax for this agreement is provided in this part of ISO/IEC 9075.

For standard programming languages C and COBOL, BOOLEAN values are mapped to integer variables in the host language. For standard programming languages Ada, Fortran, Pascal, and PL/I, BOOLEAN variables are directly supported.

For the purposes of interfacing with programming languages, the data type ARRAY shall be converted to a locator (see Subclause 4.29.5, "Locators").

For the purposes of interfacing with programming languages, the data type MULTISET shall be converted to a locator (see Subclause 4.29.5, "Locators").

For the purposes of interfacing with programming languages, user-defined types shall be handled with a locator (see Subclause 4.29.5, "Locators") or transformed to another SQL data type that has a defined mapping to the host language (see Subclause 4.7.7, "Transforms for user-defined types").

4.32 Cursors

4.32.1 General description of cursors

A cursor is a mechanism by which the rows of a table may be acted on (*e.g.*, returned to a host programming language) one at a time.

A cursor is specified by a <declare cursor>, a <dynamic declare cursor>, or an <allocate cursor statement>. A cursor specified by a <dynamic declare cursor> is a *declared dynamic cursor*. A cursor specified by an <allocate cursor statement> is an *extended dynamic cursor*. A *dynamic cursor* is either a declared dynamic cursor or an extended dynamic cursor.

For every <declare cursor> in an SQL-client module, a cursor is effectively created when an SQL-transaction (see Subclause 4.35, "SQL-transactions") referencing the SQL-client module is initiated.

For every <dynamic declare cursor> in an <SQL-client module definition>, a cursor is effectively created when an SQL-transaction (see Subclause 4.35, "SQL-transactions") referencing the <SQL-client module definition> is initiated. An extended dynamic cursor is also effectively created when an <allocate cursor statement> is executed within an SQL-session and destroyed when that SQL-session is terminated.

A dynamic cursor is destroyed when a <deallocate prepared statement> is executed that deallocates the prepared statement on which the dynamic cursor is based.

One of the properties that may be specified for a cursor determines whether or not it is a *holdable cursor*:

— A cursor that is not a holdable cursor is closed when the SQL-transaction in which it was created is terminated.

— A holdable cursor is not closed if that cursor is in the open state at the time that the SQL-transaction is terminated with a commit operation. A holdable cursor that is in the closed state at the time that the SQL-transaction is terminated remains closed. A holdable cursor is closed no matter what its state if the SQL-transaction is terminated with a rollback operation.

— A holdable cursor is closed and destroyed when the SQL-session in which it was created is terminated.

NOTE 50 — A holdable cursor may be said to be "holdable" or "held".

A cursor is in either the open state or the closed state. The initial state of a cursor is the closed state. A cursor is placed in the open state by an <open statement> and returned to the closed state by a <close statement> or a <rollback statement>. A dynamic cursor is placed in the open state by a <dynamic open statement> and

returned to the closed state by a <dynamic close statement>. An open cursor that was not defined as a holdable cursor is also closed by a <commit statement>.

A cursor in the open state identifies a table, an ordering of the rows of that table, and a position relative to that ordering. If the <declare cursor> does not contain an <order by clause>, or contains an <order by clause> that does not specify the order of the rows completely, then the rows of the table have an order that is defined only to the extent that the <order by clause> specifies an order and is otherwise implementation-dependent.

When the ordering of a cursor is not defined by an <order by clause>, the relative position of two rows is implementation-dependent. When the ordering of a cursor is partially determined by an <order by clause>, then the relative positions of two rows are determined only by the <order by clause>; if the two rows have equal values for the purpose of evaluating the <order by clause>, then their relative positions are implementation-dependent.

A cursor is either *updatable* or *not updatable*. If the table identified by a cursor is not updatable or if INSENSITIVE is specified for the cursor, then the cursor is *not updatable*; otherwise, the cursor is updatable. The operations of update and delete are permitted for updatable cursors, subject to constraining Access Rules.

The position of a cursor in the open state is either before a certain row, on a certain row, or after the last row. If a cursor is on a row, then that row is the current row of the cursor. A cursor may be before the first row or after the last row of a table even though the table is empty. When a cursor is initially opened, the position of the cursor is before the first row.

A holdable cursor that has been held open retains its position when the new SQL-transaction is initiated. However, before either an <update statement: positioned> or a <delete statement: positioned> is permitted to reference that cursor in the new SQL-transaction, a <fetch statement> shall be issued against the cursor.

4.32.2 Operations on and using cursors

A <fetch statement> positions an open cursor on a specified row of the cursor's ordering and retrieves the values of the columns of that row. An <update statement: positioned> updates the current row of the cursor. A <delete statement: positioned> deletes the current row of the cursor.

A <dynamic fetch statement> positions an open dynamic cursor on a specified row of the cursor's ordering and retrieves the values of the columns of that row. A <dynamic update statement: positioned> updates the current row of the cursor. A <dynamic delete statement: positioned> deletes the current row of the cursor.

If an error occurs during the execution of an SQL-statement that identifies a cursor, then, except where otherwise explicitly defined, the effect, if any, on the position or state of that cursor is implementation-dependent.

If a completion condition is raised during the execution of an SQL-statement that identifies a cursor, then the particular SQL-statement identifying that open cursor on which the completion condition is returned is implementation-dependent.

Another property of a cursor is its *sensitivity*, which may be sensitive, insensitive, or asensitive, depending on whether SENSITIVE, INSENSITIVE, or ASENSITIVE is specified or implied. The following paragraphs define several terms used to discuss issues relating to cursor sensitivity:

A change to SQL-data is said to be *independent* of a cursor *CR* if and only if it is not made by an <update statement: positioned> or a <delete statement: positioned> that is positioned on *CR*.

A change to SQL-data is said to be *significant* to *CR* if and only if it is independent of *CR*, and, had it been committed before *CR* was opened, would have caused the table associated with the cursor to be different in any respect.

A change to SQL-data is said to be *visible* to *CR* if and only if it has an effect on *CR* by inserting a row in *CR*, deleting a row from *CR*, changing the value of a column of a row of *CR*, or reordering the rows of *CR*.

If a cursor is open, and the SQL-transaction in which the cursor was opened makes a significant change to SQL-data, then whether that change is visible through that cursor before it is closed is determined as follows:

— If the cursor is insensitive, then significant changes are not visible.

— If the cursor is sensitive, then significant changes are visible.

— If the cursor is asensitive, then the visibility of significant changes is implementation-dependent.

If a holdable cursor is open during an SQL-transaction *T* and it is held open for a subsequent SQL-transaction, then whether any significant changes made to SQL-data (by *T* or any subsequent SQL-transaction in which the cursor is held open) are visible through that cursor in the subsequent SQL-transaction before that cursor is closed is determined as follows:

— If the cursor is insensitive, then significant changes are not visible.

— If the cursor is sensitive, then the visibility of significant changes is implementation-defined.

— If the cursor is asensitive, then the visibility of significant changes is implementation-dependent.

A <declare cursor> *DC* that specifies WITH RETURN is called a *result set cursor*. The <cursor specification> *CR* contained in *DC* defines a table *T*; the term *result set* is used to refer to *T*. A result set cursor, if declared in an SQL-invoked procedure and not closed when the procedure returns to its invoker, returns a result set to the invoker.

4.33 SQL-statements

4.33.1 Classes of SQL-statements

An SQL-statement is a string of characters that conforms to the Format and Syntax Rules specified in the parts of ISO/IEC 9075. Most SQL-statements can be prepared for execution and executed in an SQL-client module, in which case they are prepared when the SQL-client module is created and executed when the containing externally-invoked procedure is called (see Subclause 4.22, "SQL-client modules").

Most SQL-statements can be prepared for execution and executed in additional ways. These are:

— In an embedded SQL host program, in which case they are prepared when the embedded SQL host program is preprocessed (see Subclause 4.23, "Embedded syntax").

— Being prepared and executed by the use of SQL-dynamic statements (which are themselves executed in an SQL-client module or an embedded SQL host program—see Subclause 4.24, "Dynamic SQL concepts").

— Direct invocation, in which case they are effectively prepared immediately prior to execution (see Subclause 4.25, "Direct invocation of SQL").

In this part of ISO/IEC 9075, there are at least six ways of classifying SQL-statements:

— According to their effect on SQL objects, whether persistent objects, *i.e.*, SQL-data, SQL-client modules, and schemas, or transient objects, such as SQL-sessions and other SQL-statements.

— According to whether or not they start an SQL-transaction, or can, or shall, be executed when no SQL-transaction is active.

— According to whether they possibly read SQL-data or possibly modify SQL-data.

— According to whether or not they may be embedded.

— According to whether they may be dynamically prepared and executed.

— According to whether or not they may be directly executed.

This part of ISO/IEC 9075 permits SQL-implementations to provide additional, implementation-defined, statements that may fall into any of these categories. This Subclause will not mention those statements again, as their classification is implementation-defined.

The main classes of SQL-statements are:

— SQL-schema statements; these may have a persistent effect on the set of schemas.

— SQL-data statements; some of these, the SQL-data change statements, may have a persistent effect on SQL-data.

— SQL-transaction statements; except for the <commit statement>, these, and the following classes, have no effects that persist when an SQL-session is terminated.

— SQL-control statements.

— SQL-connection statements.

— SQL-session statements.

— SQL-diagnostics statements.

— SQL-dynamic statements.

— SQL embedded exception declaration.

4.33.2 SQL-statements classified by function

4.33.2.1 SQL-schema statements

The following are the SQL-schema statements:

— <schema definition>.

— <drop schema statement>.

— <domain definition>.

— <drop domain statement>.

— <table definition>.

— <drop table statement>.

— <view definition>.

— <drop view statement>.

— <assertion definition>.

— <drop assertion statement>.

— <alter table statement>.

— <alter domain statement>.

— <grant privilege statement>.

— <revoke statement>.

— <character set definition>.

— <drop character set statement>.

— <collation definition>.

— <drop collation statement>.

— <transliteration definition>.

— <drop transliteration statement>.

— <trigger definition>.

— <drop trigger statement>.

— <user-defined type definition>.

— <alter type statement>.

— <drop data type statement>.

— <user-defined ordering definition>.

— <drop user-defined ordering statement>.

— <user-defined cast definition>.

— <drop user-defined cast statement>.

— <transform definition>.

— <alter transform statement>.

— <drop transform statement>.

— <schema routine>.

— <alter routine statement>.

— <drop routine statement>.

— <sequence generator definition>.

— <alter sequence generator statement>.

— <drop sequence generator statement>.

— <role definition>.

— <grant role statement>.

— <drop role statement>.

4.33.2.2 SQL-data statements

The following are the SQL-data statements:

— <temporary table declaration>.

— <declare cursor>.

— <open statement>.

— <close statement>.

— <fetch statement>.

— <select statement: single row>.

— <free locator statement>.

— <hold locator statement>.

— <dynamic declare cursor>.

— <allocate cursor statement>.

— <dynamic select statement>.

— <dynamic open statement>.

— <dynamic close statement>.

— <dynamic fetch statement>.

— <direct select statement: multiple rows>.

— <dynamic single row select statement>.

— All SQL-data change statements.

4.33.2.3 SQL-data change statements

The following are the SQL-data change statements:

— <insert statement>.

— <delete statement: searched>.

— <delete statement: positioned>.

— <update statement: searched>.

— <update statement: positioned>.

— <merge statement>.

— <dynamic delete statement: positioned>.

— <preparable dynamic delete statement: positioned>.

— <dynamic update statement: positioned>.

— <preparable dynamic update statement: positioned>.

4.33.2.4 SQL-transaction statements

The following are the SQL-transaction statements:

— <start transaction statement>.

— <set transaction statement>.

— <set constraints mode statement>.

— <commit statement>.

— <rollback statement>.

— <savepoint statement>.

— <release savepoint statement>.

4.33.2.5 SQL-connection statements

The following are the SQL-connection statements:

— <connect statement>.

— <set connection statement>.

— <disconnect statement>.

4.33.2.6 SQL-control statements

The following are the SQL-control statements:

— <call statement>.

— <return statement>.

4.33.2.7 SQL-session statements

The following are the SQL-session statements:

— <set session characteristics statement>.

— <set session user identifier statement>.

— <set role statement>.

— <set local time zone statement>.

— <set catalog statement>.

— <set schema statement>.

— <set names statement>.

— <set path statement>.

— <set transform group statement>.

— <set session collation statement>.

4.33.2.8 SQL-diagnostics statements

The following are the SQL-diagnostics statements:

— <get diagnostics statement>.

4.33.2.9 SQL-dynamic statements

The following are the SQL-dynamic statements:

— <execute immediate statement>.

— <allocate descriptor statement>.

— <deallocate descriptor statement>.

— <get descriptor statement>.

— <set descriptor statement>.

— <prepare statement>.

— <deallocate prepared statement>.

— <describe input statement>.

— <describe output statement>.

— <execute statement>.

4.33.2.10 SQL embedded exception declaration

The following is the SQL embedded exception declaration:

— <embedded exception declaration>.

4.33.3 SQL-statements and SQL-data access indication

Some SQL-statements may be classified either as SQL-statements that *possibly read SQL-data* or that *possibly modify SQL-data*. A given SQL-statement belongs to at most one such class.

The following SQL-statements possibly read SQL-data:

— SQL-data statements other than SQL-data change statements, <free locator statement>, and <hold locator statement>.

— SQL-statements that simply contain a <subquery> and that are not SQL-statements that possibly modify SQL-data.

The following SQL-statements possibly modify SQL-data:

— SQL-schema statements.

— SQL-data change statements.

4.33.4 SQL-statements and transaction states

The following SQL-statements are transaction-initiating SQL-statements, *i.e.*, if there is no current SQL-transaction, and a statement of this class is executed, an SQL-transaction is initiated:

— All SQL-schema statements

— The SQL-transaction statements <commit statement> and <rollback statement>, if they specify AND CHAIN.

— The following SQL-data statements:

- <open statement>.
- <close statement>.
- <fetch statement>.
- <select statement: single row>.
- <insert statement>.
- <delete statement: searched>.
- <delete statement: positioned>.
- <update statement: searched>.
- <update statement: positioned>.
- <merge statement>.
- <allocate cursor statement>.
- <dynamic open statement>.
- <dynamic close statement>.
- <dynamic fetch statement>.
- <direct select statement: multiple rows>.
- <dynamic single row select statement>.
- <dynamic delete statement: positioned>.
- <preparable dynamic delete statement: positioned>.
- <dynamic update statement: positioned>.
- <preparable dynamic update statement: positioned>.
- <free locator statement>.
- <hold locator statement>.

— <start transaction statement>.

— The following SQL-dynamic statements:

- <describe input statement>.
- <describe output statement>.
- <allocate descriptor statement>.
- <deallocate descriptor statement>.
- <get descriptor statement>.
- <set descriptor statement>.
- <prepare statement>.
- <deallocate prepared statement>.

Whether or not an <execute immediate statement> starts a transaction depends on the content of the <SQL statement variable> referenced by the <execute immediate statement> at the time it is executed. Whether or not an <execute statement> starts a transaction depends on the content of the <SQL statement variable> referenced by the <prepare statement> at the time the prepared statement referenced by the <execute statement> was prepared. In both cases, if the content of the <SQL statement variable> was a transaction-initiating SQL-statement, then the <execute immediate statement> or <execute statement> is treated as a transaction-initiating statement; otherwise it is not treated as a transaction-initiating statement.

The following SQL-statements are not transaction-initiating SQL-statements, *i.e.*, if there is no current SQL-transaction, and a statement of this class is executed, no SQL-transaction is initiated.

— All SQL-transaction statements except <start transaction statement>s and <commit statement>s and <rollback statement>s that specify AND CHAIN.

— All SQL-connection statements.

— All SQL-session statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <temporary table declaration>.
- <declare cursor>.
- <dynamic declare cursor>.
- <dynamic select statement>.

The following SQL-statements are possibly transaction-initiating SQL-statements:

— <return statement>.

— <call statement>.

If the initiation of an SQL-transaction occurs in an atomic execution context, and an SQL-transaction has already completed in this context, then an exception condition is raised: *invalid transaction initiation*.

If an <SQL control statement> causes the evaluation of a <subquery> and there is no current SQL-transaction, then an SQL-transaction is initiated before evaluation of the <subquery>.

4.33.5 SQL-statement atomicity and statement execution contexts

The execution of all SQL-statements other than certain SQL-control statements and certain SQL-transaction statements is atomic with respect to recovery. Such an SQL-statement is called an *atomic SQL-statement*. An SQL-statement that is not an atomic SQL-statement is called a *non-atomic SQL statement*.

The following are non-atomic SQL-statements:

— <call statement>

— <execute statement>

— <execute immediate statement>

— <commit statement>

— <return statement>

— <rollback statement>

— <savepoint statement>

All other SQL-statements are atomic SQL-statements.

A statement execution context is either *atomic* or *non-atomic*.

The statement execution context brought into existence by the execution of a non-atomic SQL-statement is a *non-atomic execution context*.

The statement execution context brought into existence by the execution of an atomic SQL-statement or the evaluation of a <subquery> is an *atomic execution context*.

Within one execution context, another execution context may become active. This latter execution context is said to be a *more recent execution context* than the former. If there is no execution context that is more recent than execution context *EC*, then *EC* is said to be the *most recent execution context*.

If there is no atomic execution context that is more recent than atomic execution context *AEC*, then *AEC* is the *most recent atomic execution context*.

An SQL-transaction cannot be explicitly terminated within an atomic execution context. If the execution of an atomic SQL-statement is unsuccessful, then the changes to SQL-data or schemas made by the SQL-statement are canceled.

4.33.6 Embeddable SQL-statements

The following SQL-statements are embeddable in an embedded SQL host program, and may be the <SQL procedure statement> in an <externally-invoked procedure> in an <SQL-client module definition>:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-connection statements.

— All SQL-session statements.

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— The following SQL-data statements:

- <allocate cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <insert statement>.
- <delete statement: searched>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <update statement: searched>.
- <update statement: positioned>.
- <merge statement>.
- <dynamic update statement: positioned>.
- <hold locator statement>.
- <free locator statement>.

— The following SQL-control statements:

- <call statement>.
- <return statement>.

The following SQL-statements are embeddable in an embedded SQL host program, and may occur in an <SQL-client module definition>, though not in an <externally-invoked procedure>:

— <temporary table declaration>.

— <declare cursor>.

— <dynamic declare cursor>.

The following SQL-statements are embeddable in an embedded SQL host program, but may not occur in an <SQL-client module definition>:

— SQL embedded exception declarations.

Consequently, the following SQL-data statements are not embeddable in an embedded SQL host program, nor may they occur in an <SQL-client module definition>, nor be the <SQL procedure statement> in an <externally-invoked procedure> in an <SQL-client module definition>:

— <dynamic select statement>.

— <dynamic single row select statement>.

— <direct select statement: multiple rows>.

— <preparable dynamic delete statement: positioned>.

— <preparable dynamic update statement: positioned>.

4.33.7 Preparable and immediately executable SQL-statements

The following SQL-statements are preparable:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-session statements.

— The following SQL-data statements:

- <delete statement: searched>.
- <dynamic select statement>.
- <dynamic single row select statement>.
- <insert statement>.
- <update statement: searched>.
- <merge statement>.
- <preparable dynamic delete statement: positioned>.
- <preparable dynamic update statement: positioned>.
- <preparable implementation-defined statement>.

- <hold locator statement>.
- <free locator statement>.

— The following SQL-control statements:

- <call statement>.

Consequently, the following SQL-statements are not preparable:

— All SQL-connection statements.

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <allocate cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <update statement: positioned>.
- <dynamic update statement: positioned>.
- <direct select statement: multiple rows>.
- <temporary table declaration>.
- <declare cursor>.
- <dynamic declare cursor>.

— The following SQL-control statements:

- <return statement>.

Any preparable SQL-statement can be executed immediately, with the exception of:

— <dynamic select statement>.

— <dynamic single row select statement>.

4.33.8 Directly executable SQL-statements

The following SQL-statements may be executed directly:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-connection statements.

— All SQL-session statements.

— The following SQL-data statements:

- <temporary table declaration>.
- <direct select statement: multiple rows>.
- <insert statement>.
- <delete statement: searched>.
- <update statement: searched>.
- <merge statement>.

— The following SQL-control statements:

- <call statement>.
- <return statement>.

Consequently, the following SQL-statements may not be executed directly:

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <declare cursor>.
- <dynamic declare cursor>.
- <allocate cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.

- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <dynamic select statement>.
- <dynamic single row select statement>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <preparable dynamic delete statement: positioned>.
- <update statement: positioned>.
- <dynamic update statement: positioned>.
- <preparable dynamic update statement: positioned>.

— <free locator statement>.

— <hold locator statement>.

4.34 Basic security model

4.34.1 Authorization identifiers

An <authorization identifier> identifies a set of privileges. An <authorization identifier> is either a user identifier or a role name. A user identifier represents a user of the database system. The mapping of user identifiers to operating system users is implementation-dependent. A role name represents a role.

4.34.1.1 SQL-session authorization identifiers

An SQL-session has a <user identifier> called the *SQL-session user identifier*. When an SQL-session is initiated, the SQL-session user identifier is determined in an implementation-defined manner, unless the session is initiated using a <connect statement>. The value of the SQL-session user identifier can never be the null value. The SQL-session user identifier can be determined by using SESSION_USER.

An SQL-session context contains a time-varying sequence of cells, known as the *authorization stack*, each cell of which contains either a user identifier, a role name, or both. This stack is maintained using a "last-in, first-out" discipline, and effectively only the top cell is visible. When an SQL-session is started, by explicit or implicit execution of a <connect statement>, the authorization stack is initialized with one cell, which contains only the

user identifier known as the *SQL-session user identifier*; a role name, known as the *SQL-session role name* may be added subsequently.

Let *E* be an externally-invoked procedure, SQL-invoked routine, triggered action, prepared statement, or directly executed statement. When *E* is invoked, a copy of the top cell is pushed onto the authorization stack. If the invocation of *E* is to be under definer's rights, then the contents of the top cell are replaced with the authorization identifier of the owner of *E*. On completion of the execution of *E*, the top cell is removed.

The contents of the top cell in the authorization stack of the current SQL-session context determine the privileges for the execution of each SQL-statement. The user identifier, if any, in this cell is known as the *current user identifier*; the role name, if any, is known as the *current role name*. They may be determined using CURRENT_USER and CURRENT_ROLE, respectively.

At a given time, there may be no current user identifier or no current role name, but at least one or the other is always present.

NOTE 51 — The privileges granted to PUBLIC are available to all of the <authorization identifier>s in the SQL-environment.

The <set session user identifier statement> changes the value of the current user identifier and of the SQL-session user identifier. The <set role statement> changes the value of the current role name.

The term *current authorization identifier* denotes an authorization identifier in the top cell of the authorization stack.

4.34.1.2 SQL-client module authorization identifiers

If an <SQL-client module definition> contains a <module authorization identifier> *MAI*, then *MAI* is the owner of the corresponding SQL-client module *M* and is used as the current authorization identifier for the execution of each externally-invoked procedure in *M*. If *M* has no owner, then the current user identifier and the current role name of the SQL-session are used as the current user identifier and current role name, respectively, for the execution of each externally-invoked procedure in *M*.

4.34.1.3 SQL-schema authorization identifiers

Every schema has an owner, determined at the time of its creation from a <schema definition> *SD*. That owner is

Case:

— If *SD* simply contains a <schema authorization identifier> *SAI*, then *SAI*.

— If *SD* is simply contained in an <SQL-client module definition> that contains a <module authorization identifier> *MAI*, then *MAI*.

— Otherwise, the SQL-session user identifier.

4.34.2 Privileges

A privilege authorizes a given category of <action> to be performed on a specified base table, view, column, domain, character set, collation, transliteration, user-defined type, trigger, SQL-invoked routine, or sequence generator by a specified <authorization identifier>.

Each privilege is represented by a *privilege descriptor*. A privilege descriptor contains:

— The identification of the base table, view, column, domain, character set, collation, transliteration, user-defined type, table/method pair, trigger, SQL-invoked routine, or sequence generator that the descriptor describes.

— The <authorization identifier> of the grantor of the privilege.

— The <authorization identifier> of the grantee of the privilege.

— Identification of the <action> that the privilege allows.

— An indication of whether or not the privilege is grantable.

— An indication of whether or not the privilege has the WITH HIERARCHY OPTION specified.

The <action>s that can be specified are:

— INSERT

— INSERT (<column name list>)

— UPDATE

— UPDATE (<column name list>)

— DELETE

— SELECT

— SELECT (<column name list>)

— SELECT (<privilege method list>)

— REFERENCES

— REFERENCES (<column name list>)

— USAGE

— UNDER

— TRIGGER

— EXECUTE

A privilege descriptor with an <action> of INSERT, UPDATE, DELETE, SELECT, TRIGGER, or REFERENCES is called a *table privilege descriptor* and identifies the existence of a privilege on the table identified by the privilege descriptor.

A privilege descriptor with an <action> of SELECT (<column name list>), INSERT (<column name list>), UPDATE (<column name list>), or REFERENCES (<column name list>) is called a *column privilege descriptor* and identifies the existence of a privilege on the columns in the table identified by the privilege descriptor.

A privilege descriptor with an <action> of SELECT (<privilege method list>) is called a *table/method privilege descriptor* and identifies the existence of a privilege on the methods of the structured type of the table identified by the privilege descriptor.

A table privilege descriptor specifies that the privilege identified by the <action> (unless the <action> is DELETE) is to be automatically granted by the grantor to the grantee on all columns subsequently added to the table.

A privilege descriptor with an <action> of USAGE is called a *usage privilege descriptor* and identifies the existence of a privilege on the domain, user-defined type, character set, collation, transliteration, or sequence generator identified by the privilege descriptor.

A privilege descriptor with an <action> of UNDER is called an *under privilege descriptor* and identifies the existence of the privilege on the structured type identified by the privilege descriptor.

A privilege descriptor with an <action> of EXECUTE is called an *execute privilege descriptor* and identifies the existence of a privilege on the SQL-invoked routine identified by the privilege descriptor.

A grantable privilege is a privilege associated with a schema that may be granted by a <grant statement>. The WITH GRANT OPTION clause of a <grant statement> specifies whether the <authorization identifier> recipient of a privilege (acting as a grantor) may grant it to others.

Privilege descriptors that represent privileges for the owner of an object have a special grantor value, "_SYSTEM". This value is reflected in the Information Schema for all privileges that apply to the owner of the object.

NOTE 52 — The Information Schema is defined in ISO/IEC 9075-11.

A schema that is owned by a given schema <user identifier> or schema <role name> may contain privilege descriptors that describe privileges granted to other <authorization identifier>s (grantees). The granted privileges apply to objects defined in the current schema.

Direct SQL statements are always executed under invoker's rights.

4.34.3 Roles

A role, identified by a <role name>, is a set of privileges defined by the union of the privileges defined by the privilege descriptors whose grantee is that <role name> and the sets of privileges for the <role name>s defined by the role authorization descriptors whose grantee is the first <role name>. A role may be granted to <authorization identifier>s with a <grant role statement>. No cycles of role grants are allowed.

The WITH ADMIN OPTION clause of the <grant role statement> specifies whether the recipient of a role may grant it to others.

Each grant is represented and identified by a *role authorization descriptor*. A role authorization descriptor includes:

— The role name of the role.

— The <authorization identifier> of the grantor.

— The <authorization identifier> of the grantee.

— An indication of whether or not the role was granted with the WITH ADMIN OPTION and hence is grantable.

Because roles may be granted to other roles, a role is said to "contain" other roles. The set of roles *X* contained in any role *A* is defined as the set of roles identified by role authorization descriptors whose grantee is *A*, together with all other roles contained by roles in *X*.

4.34.4 Security model definitions

The term *enabled authorization identifiers* denotes the set of authorization identifiers whose members are the current user identifier, the current role name, and every role name that is contained in the current role name.

The term *applicable privileges* for an authorization identifier *A* denotes the union of the set of privileges whose grantee is PUBLIC with the set of privileges whose grantees are *A* and, if *A* is a role name, every role name contained in *A*.

The term *current privileges* denotes the union of the applicable privileges for the current user identifier with the applicable privileges for the current role name.

4.35 SQL-transactions

4.35.1 General description of SQL-transactions

An *SQL-transaction* (transaction) is a sequence of executions of SQL-statements that is atomic with respect to recovery. These operations are performed by one or more compilation units and SQL-client modules. The operations comprising an SQL-transaction may also be performed by the direct invocation of SQL.

It is implementation-defined whether or not the execution of an SQL-data statement is permitted to occur within the same SQL-transaction as the execution of an SQL-schema statement. If it does occur, then the effect on any open cursor or deferred constraint is implementation-defined. There may be additional implementation-defined restrictions, requirements, and conditions. If any such restrictions, requirements, or conditions are violated, then an implementation-defined exception condition or a completion condition *warning* with an implementation-defined subclass code is raised.

It is implementation-defined whether or not the dynamic execution of an <SQL dynamic data statement> is permitted to occur within the same SQL-transaction as the dynamic execution of an SQL-schema statement. If it does occur, then the effect on any open cursor, prepared dynamic statement, or deferred constraint is implementation-defined. There may be additional implementation-defined restrictions, requirements, and conditions. If any such restrictions, requirements, or conditions are violated, then an implementation-defined exception condition or a completion condition *warning* with an implementation-defined subclass code is raised.

Each SQL-client module that executes an SQL-statement of an SQL-transaction is associated with that SQL-transaction. Each direct invocation of SQL that executes an SQL-statement of an SQL-transaction is associated with that SQL-transaction. An SQL-transaction is initiated when no SQL-transaction is currently active by direct invocation of SQL that results in the execution of a transaction-initiating <direct SQL statement>. An SQL-transaction is initiated when no SQL-transaction is currently active and an <externally-invoked procedure> is called that results in the execution of a *transaction-initiating* SQL-statement. An SQL-transaction is terminated by a <commit statement> or a <rollback statement>. If an SQL-transaction is terminated by successful execution of a <commit statement>, then all changes made to SQL-data or schemas by that SQL-transaction are made persistent and accessible to all concurrent and subsequent SQL-transactions. If an SQL-transaction is terminated by a <rollback statement> or unsuccessful execution of a <commit statement>, then all changes made to SQL-data or schemas by that SQL-transaction are canceled. Committed changes cannot be canceled. If execution of a <commit statement> is attempted, but certain exception conditions are raised, it is unknown whether or not the changes made to SQL-data or schemas by that SQL-transaction are canceled or made persistent.

4.35.2 Savepoints

An SQL-transaction may be partially rolled back by using a savepoint. The savepoint and its <savepoint name> are established within an SQL-transaction when a <savepoint statement> is executed.

An SQL-transaction has one or more *savepoint levels*, exactly one of which is the *current savepoint level*. The savepoint levels of an SQL-transaction are nested, such that when a *new savepoint level NSL is established*, the current savepoint level *CSL* ceases to be current and *NSL* becomes current. When *NSL is destroyed*, *CSL* becomes current again.

A savepoint level exists in an SQL-session *SS* even when no SQL-transaction is active, this savepoint level remaining the current one when an SQL-transaction is initiated in *SS*.

A savepoint *SP* exists at exactly one savepoint level, namely, the savepoint level that is current when *SP* is established.

If a <rollback statement> references a savepoint *SS*, then all changes made to SQL-data or schema subsequent to the establishment of the savepoint are canceled, all savepoints established since *SS* was established are destroyed, and the SQL-transaction is restored to its state as it was immediately following the execution of the <savepoint statement>. Savepoints existing at savepoint level *SPL* are destroyed when *SPL* is destroyed. Savepoint *SS* in the current savepoint level and all savepoints established since *SS* was established are destroyed when a <release savepoint statement> specifying the savepoint name of *SS* is executed. A savepoint may be replaced by another with the same name within a savepoint level by executing a <savepoint statement> that specifies that name.

It is implementation-defined whether or not, or how, a <rollback statement> that references a <savepoint specifier> affects diagnostics area contents, the contents of SQL descriptor areas, and the status of prepared statements.

4.35.3 Properties of SQL-transactions

An SQL-transaction has a *constraint mode* for each integrity constraint. The constraint mode for an integrity constraint in an SQL-transaction is described in Subclause 4.17, "Integrity constraints".

An SQL-transaction has an *access mode* that is either *read-only* or *read-write*. The access mode may be explicitly set by a <set transaction statement> before the start of an SQL-transaction or by the use of a <start transaction statement> to start an SQL-transaction; otherwise, it is implicitly set to the default access mode for the SQL-session before each SQL-transaction begins. If no <set session characteristics statement> has set the default access mode for the SQL-session, then the default access mode for the SQL-session is *read-write*. The term *read-only* applies only to viewed tables and persistent base tables.

An SQL-transaction has a *condition area limit*, which is a positive integer that specifies the maximum number of conditions that can be placed in any diagnostics area during execution of an SQL-statement in this SQL-transaction.

SQL-transactions initiated by different SQL-agents that access the same SQL-data or schemas and overlap in time are *concurrent SQL-transactions*.

4.35.4 Isolation levels of SQL-transactions

An SQL-transaction has an *isolation level* that is READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE. The isolation level of an SQL-transaction defines the degree to which the operations on SQL-data or schemas in that SQL-transaction are affected by the effects of and can affect operations on SQL-data or schemas in concurrent SQL-transactions. The isolation level of an SQL-transaction when any cursor is held open from the previous SQL-transaction within an SQL-session is the isolation level of the previous SQL-transaction by default. If no cursor is held open, or this is the first SQL-transaction within an SQL-session, then the isolation level is SERIALIZABLE by default. The level can be explicitly set by the <set transaction statement> before the start of an SQL-transaction or by the use of a <start transaction statement> to start an SQL-transaction. If it is not explicitly set, then the isolation level is implicitly set to the default isolation level for the SQL-session before each SQL-transaction begins. If no <set session characteristics statement> has set the default isolation level for the SQL-session, then the default isolation level for the SQL-session is SERIALIZABLE.

Execution of a <set transaction statement> is prohibited after the start of an SQL-transaction and before its termination. Execution of a <set transaction statement> before the start of an SQL-transaction sets the access mode, isolation level, and condition area limit for the single SQL-transaction that is started after the execution of that <set transaction statement>. If multiple <set transaction statement>s are executed before the start of an SQL-transaction, the last such statement is the one whose settings are effective for that SQL-transaction; their actions are not cumulative.

The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.

The isolation level specifies the kind of phenomena that can occur during the execution of concurrent SQL-transactions. The following phenomena are possible:

1) *P1* ("Dirty read"): SQL-transaction *T1* modifies a row. SQL-transaction *T2* then reads that row before *T1* performs a COMMIT. If *T1* then performs a ROLLBACK, *T2* will have read a row that was never committed and that may thus be considered to have never existed.

2) *P2* ("Non-repeatable read"): SQL-transaction *T1* reads a row. SQL-transaction *T2* then modifies or deletes that row and performs a COMMIT. If *T1* then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.

3) *P3* ("Phantom"): SQL-transaction *T1* reads the set of rows *N* that satisfy some <search condition>. SQL-transaction *T2* then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction *T1*. If SQL-transaction *T1* then repeats the initial read with the same <search condition>, it obtains a different collection of rows.

The four isolation levels guarantee that each SQL-transaction will be executed completely or not at all, and that no updates will be lost. The isolation levels are different with respect to phenomena *P1*, *P2*, and *P3*. Table 8, "SQL-transaction isolation levels and the three phenomena" specifies the phenomena that are possible and not possible for a given isolation level.

Table 8 — SQL-transaction isolation levels and the three phenomena

Level	*P1*	*P2*	*P3*
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

NOTE 53 — The exclusion of these phenomena for SQL-transactions executing at isolation level SERIALIZABLE is a consequence of the requirement that such transactions be serializable.

Changes made to SQL-data or schemas by an SQL-transaction in an SQL-session may be perceived by that SQL-transaction in that same SQL-session, and by other SQL-transactions, or by that same SQL-transaction in other SQL-sessions, at isolation level READ UNCOMMITTED, but cannot be perceived by other SQL-transactions at isolation level READ COMMITTED, REPEATABLE READ, or SERIALIZABLE until the former SQL-transaction terminates with a <commit statement>.

Regardless of the isolation level of the SQL-transaction, phenomena *P1*, *P2*, and *P3* shall not occur during the implied reading of schema definitions performed on behalf of executing an SQL-statement, the checking of integrity constraints, and the execution of referential actions associated with referential constraints. The schema definitions that are implicitly read are implementation-dependent. This does not affect the explicit reading of rows from tables in the Information Schema, which is done at the isolation level of the SQL-transaction.

NOTE 54 — The Information Schema is defined in ISO/IEC 9075-11.

4.35.5 Implicit rollbacks

The execution of a <rollback statement> may be initiated implicitly by an SQL-implementation when it detects the inability to guarantee the serializability of two or more concurrent SQL-transactions. When this error occurs, an exception condition is raised: *transaction rollback — serialization failure*.

The execution of a <rollback statement> may be initiated implicitly by an SQL-implementation when it detects unrecoverable errors. When such an error occurs, an exception condition is raised: *transaction rollback* with an implementation-defined subclass code.

4.35.6 Effects of SQL-statements in an SQL-transaction

The execution of an SQL-statement within an SQL-transaction has no effect on SQL-data or schemas other than the effect stated in the General Rules for that SQL-statement, in the General Rules for Subclause 11.8, "<referential constraint definition>", in the General Rules for Subclause 11.39, "<trigger definition>", and in the General Rules for Subclause 11.50, "<SQL-invoked routine>". Together with serializable execution, this implies that all read operations are repeatable within an SQL-transaction at isolation level SERIALIZABLE, except for:

1) The effects of changes to SQL-data or schemas and its contents made explicitly by the SQL-transaction itself.

2) The effects of differences in SQL parameter values supplied to externally-invoked procedures.

3) The effects of references to time-varying system variables such as CURRENT_DATE and CURRENT_USER.

4.35.7 Encompassing transactions

In some environments (*e.g.*, remote database access), an SQL-transaction can be part of an encompassing transaction that is controlled by an agent other than the SQL-agent. The encompassing transaction may involve different resource managers, the SQL-implementation being just one instance of such a manager. In such environments, an encompassing transaction shall be terminated via that other agent, which in turn interacts with the SQL-implementation via an interface that may be different from SQL (COMMIT or ROLLBACK), in order to coordinate the orderly termination of the encompassing transaction. When an SQL-transaction is part of an encompassing transaction that is controlled by an agent other than an SQL-agent and a <rollback statement> is initiated implicitly by an SQL-implementation, then the SQL-implementation will interact with that other agent to terminate that encompassing transaction. The specification of the interface between such agents and the SQL-implementation is beyond the scope of this part of ISO/IEC 9075. However, it is important to note that the semantics of an SQL-transaction remain as defined in the following sense:

— When an agent that is different from the SQL-agent requests the SQL-implementation to rollback an SQL-transaction, the General Rules of Subclause 16.7, "<rollback statement>", are performed.

— When such an agent requests the SQL-implementation to commit an SQL-transaction, the General Rules of Subclause 16.6, "<commit statement>", are performed. To guarantee orderly termination of the encompassing transaction, this commit operation may be processed in several phases not visible to the application; not all the General Rules of Subclause 16.6, "<commit statement>", need to be executed in a single phase.

However, even in such environments, the SQL-agent interacts directly with the SQL-server to set characteristics (such as *read-only* or *read-write*, isolation level, and constraints mode) that are specific to the SQL-transaction model.

It is implementation-defined whether SQL-transactions that affect more than one SQL-server are supported. If such SQL-transactions are supported, then the part of each SQL-transaction that affects a single SQL-server is called a *branch transaction* or a branch of the SQL-transaction. If such SQL-transactions are supported, then they generally have all the same characteristics (access mode, condition area limit, and isolation level, as well as constraint mode). However, it is possible to alter some characteristics of such an SQL-transaction at one SQL-server by the use of the SET LOCAL TRANSACTION statement; if a SET LOCAL TRANSACTION statement is executed at an SQL-server before any transaction-initiating SQL-statement, then it may set the characteristics of that *branch* of the SQL-transaction at that SQL-server.

The characteristics of a branch of an SQL-transaction are limited by the characteristics of the SQL-transaction as a whole:

— If the SQL-transaction is read-write, then the branch of the SQL-transaction may be read-write or read-only; if the SQL-transaction is read-only, then the branch of the SQL-transaction shall be read-only.

— If the SQL-transaction has an isolation level of READ UNCOMMITTED, then the branch of the SQL-transaction may have an isolation level of READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

  If the SQL-transaction has an isolation level of READ COMMITTED, then the branch of the SQL-transaction shall have an isolation level of READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

  If the SQL-transaction has an isolation level of REPEATABLE READ, then the branch of the SQL-transaction shall have an isolation level of REPEATABLE READ or SERIALIZABLE.

  If the SQL-transaction has an isolation level of SERIALIZABLE, then the branch of the SQL-transaction shall have an isolation level of SERIALIZABLE.

— The diagnostics area limit of a branch of an SQL-transaction is always the same as the condition area limit of the SQL-transaction; SET LOCAL TRANSACTION shall not specify a condition area limit.

SQL-transactions that are not part of an encompassing transaction are terminated by the execution of <commit statement>s and <rollback statement>s. If those statements specify AND CHAIN, then they also initiate a new SQL-transaction with the same characteristics as the SQL-transaction that was just terminated, except that the constraint mode of each integrity constraint reverts to its default mode (*deferred* or *immediate*).

4.36 SQL-connections

An *SQL-connection* is an association between an SQL-client and an SQL-server. An SQL-connection may be established and named by a <connect statement>, which identifies the desired SQL-server by means of an <SQL-server name>. A <connection name> is specified as a <simple value specification> whose value is an <identifier>. Two <connection name>s identify the same SQL-connection if their values, with leading and trailing <space>s removed, are equivalent according to the rules for <identifier> comparison in Subclause 5.2, "<token> and <separator>". It is implementation-defined how an SQL-implementation uses <SQL-server name> to determine the location, identity, and communication protocol required to access the SQL-server and create an SQL-session.

An SQL-connection is an *active SQL-connection* if any SQL-statement that initiates or requires an SQL-transaction has been executed at its SQL-server via that SQL-connection during the current SQL-transaction.

An SQL-connection is either *current* or *dormant*. If the SQL-connection established by the most recently executed implicit or explicit <connect statement> or <set connection statement> has not been terminated, then that SQL-connection is the *current SQL-connection*; otherwise, there is no current SQL-connection. An existing SQL-connection that is not the current SQL-connection is a *dormant SQL-connection*.

An SQL implementation may detect the loss of the current SQL-connection during execution of any SQL-statement. When such a connection failure is detected, an exception condition is raised: *transaction rollback — statement completion unknown*. This exception condition indicates that the results of the actions performed in the SQL-server on behalf of the statement are unknown to the SQL-agent.

Similarly, an SQL-implementation may detect the loss of the current SQL-connection during the execution of a <commit statement>. When such a connection failure is detected, an exception condition is raised: *connection exception — transaction resolution unknown*. This exception condition indicates that the SQL-implementation cannot verify whether the SQL-transaction was committed successfully, rolled back, or left active.

A user may initiate an SQL-connection between the SQL-client associated with the SQL-agent and a specific SQL-server by executing a <connect statement>. Otherwise, an SQL-connection between the SQL-client and an implementation-defined default SQL-server is initiated when an externally-invoked procedure is called and no SQL-connection is current. The SQL-connection associated with an implementation-defined default SQL-server is called the *default SQL-connection*. An SQL-connection is terminated either by executing a <disconnect statement>, or following the last call to an externally-invoked procedure within the last active SQL-client module, or by the last execution of a <direct SQL statement> through the direct invocation of SQL. The mechanism and rules by which an SQL-implementation determines whether a call to an externally-invoked procedure is the last call within the last active SQL-client module and the mechanism and rules by which an SQL-implementation determines whether a direct invocation of SQL is the last execution of a <direct SQL statement> are implementation-defined.

An SQL-implementation shall support at least one SQL-connection and may require that the SQL-server be identified at the binding time chosen by the SQL-implementation. If an SQL-implementation permits more than one concurrent SQL-connection, then the SQL-agent may connect to more than one SQL-server and select the SQL-server by executing a <set connection statement>.

4.37 SQL-sessions

4.37.1 General description of SQL-sessions

An *SQL-session* spans the execution of a sequence of consecutive SQL-statements invoked either by a single user from a single SQL-agent or by the direct invocation of SQL. At any one time during an SQL-session, exactly one of the SQL-statements in this sequence is being executed and is said to be an *executing statement*. In some cases, an executing statement *ES* causes a nested sequence of consecutive SQL-statements to be executed as a direct result of *ES*; during that time, exactly one of these is also an executing statement and it in turn might similarly involve execution of a further nested sequence, and so on, indefinitely. An executing statement *ES* such that no statement is executing as a direct result of *ES* is called the *innermost executing statement* of the SQL-session.

An SQL-session is associated with an SQL-connection. The SQL-session associated with the default SQL-connection is called the *default SQL-session*. An SQL-session is either *current* or *dormant*. The *current SQL-*

session is the SQL-session associated with the current SQL-connection. A *dormant SQL-session* is an SQL-session that is associated with a dormant SQL-connection.

Within an SQL-session, module local temporary tables are effectively created by <temporary table declaration>s. Module local temporary tables are accessible only to invocations of <externally-invoked procedure>s in the SQL-client module in which they are created. The definitions of module local temporary tables persist until the end of the SQL-session.

Within an SQL-session, locators are effectively created when a host parameter, a host variable, or an SQL parameter of an external routine that is specified as a binary large object locator, a character large object locator, a user-defined type locator, an array locator, or a multiset locator is assigned a value of binary large object type, character large object type, user-defined type, array type, or multiset type, respectively. These locators are part of the SQL-session context. A locator may be either valid or invalid. All locators remaining valid at the end of an SQL-session are marked invalid on termination of that SQL-session. A host variable that is a locator may be *holdable* or *nonholdable*.

4.37.2 SQL-session identification

An SQL-session has a unique implementation-dependent SQL-session identifier. This SQL-session identifier is different from the SQL-session identifier of any other concurrent SQL-session. The SQL-session identifier is used to effectively define implementation-defined schemas that contain the instances of any global temporary tables, created local temporary tables, or declared local temporary tables within the SQL-session.

An SQL-session is started as a result of successful execution of a <connect statement>, which sets the initial SQL-session user identifier to the value of the implicit or explicit <connection user name> contained in the <connect statement>.

An SQL-session initially has no SQL-session role name.

An SQL-session has an original time zone displacement and a current default time zone displacement, which are values of data type INTERVAL HOUR TO MINUTE. Both the original time zone displacement and the current default time zone displacement are initially set to the same implementation-defined value. The current default time zone displacement can subsequently be changed by successful execution of a <set local time zone statement>. The original time zone displacement cannot be changed. It is also possible to set the current default time zone displacement to equal the value of the original time zone displacement.

An SQL-session has a default catalog name that is used to effectively qualify unqualified <schema name>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The default catalog name is initially set to an implementation-defined value but can subsequently be changed by the successful execution of a <set catalog statement> or <set schema statement>.

An SQL-session has a default unqualified schema name that is used to effectively qualify unqualified <schema qualified name>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The default unqualified schema name is initially set to an implementation-defined value but can subsequently be changed by the successful execution of a <set schema statement>.

4.37.3 SQL-session properties

An SQL-session has an SQL-path that is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The SQL-path is initially set to an implementation-defined value, but can subsequently be changed by the successful execution of a <set path statement>.

The text defining the SQL-path can be referenced by using the <general value specification> CURRENT_PATH.

An SQL-session has a default transform group name and one or more user-defined type name—transform group name pairs that are used to identify the group of transform functions for every user-defined type that is referenced in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The transform group name for a given user-defined type name is initially set to an implementation-defined value but can subsequently be changed by the successful execution of a <set transform group statement>.

The text defining the transform group names associated with the SQL-session can be referenced using two mechanisms: the <general value specification> "CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>", which evaluates to the name of the transform group associated with the specified data type, and the <general value specification> "CURRENT_DEFAULT_TRANSFORM_GROUP", which evaluates to the name of the transform group associated with all types that have no type-specific transform group specified for them.

An SQL-session has a default character set name that is used to identify the character set in which <preparable statement>s are represented when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement>. The default character set name is initially set to an implementation-defined value but can subsequently be changed by the successful execution of a <set names statement>.

For each character set known to the SQL-implementation, an SQL-session has at most one SQL-session collation for that character set, to be used when the rules of Subclause 9.13, "Collation determination", are applied. There are no SQL-session collations at the start of an SQL-session. The SQL-session collation for a character set can be set or changed by the successful execution of a <set session collation statement>.

An SQL-invoked routine is *active* as soon as an SQL-statement executed by an SQL-agent causes invocation of an SQL-invoked routine and ceases to be active when execution of that invocation is complete.

At any time during an SQL-session, *containing SQL* is said to be *permitted* or *not permitted*. Similarly, *reading SQL-data* is said to be *permitted* or *not permitted* and *modifying SQL-data* is said to be *permitted* or *not permitted*.

An SQL-session has *enduring characteristics*. The enduring characteristics of an SQL-session are initially the same as the default values for the corresponding SQL-session characteristics. The enduring characteristics are changed by successful execution of a <set session characteristics statement> that specifies one or more enduring characteristics. Enduring characteristics that are not specified in a <set session characteristics statement> are not changed in any way by the successful execution of that statement.

SQL-sessions have the following enduring characteristics:

— *enduring transaction characteristics*

Each of the enduring characteristics are affected by a corresponding alternative in the <session characteristic> appearing in the <session characteristic list> of a <set session characteristics statement>.

An SQL-session has a stack of contexts that is preserved when an SQL-session is made dormant and restored when the SQL-session is made active. Each context in the stack comprises:

— The SQL-session identifier.

— The authorization stack.

— The identities of all instances of temporary tables.

— The original time zone displacement.

— The current default time zone displacement.

— The current constraint mode for each integrity constraint.

— The current transaction access mode.

— The cursor position of all open cursors.

— The current transaction isolation level.

— The current SQL diagnostics area stack and its contents, along with the current condition area limit.

— The value of all valid locators.

— The value of the SQL-path for the current SQL-session.

— A *statement execution context*.

— A *routine execution context*.

— Zero or more *trigger execution contexts*.

— All prepared statements prepared during the current SQL-session and not deallocated.

— The current default catalog name.

— The current default unqualified schema name.

— The current default character set name.

— For each character set known to the SQL-implementation, the SQL-session collation, if any.

— The text defining the SQL-path.

— The contents of all SQL dynamic descriptor areas.

— The text defining the default transform group name.

— The text defining the user-defined type name—transform group name pair for each user-defined type explicitly set by the user.

NOTE 55 — The use of the word "current" in the preceding list implies the values that are current in the SQL-session that is to be made dormant, and not the values that will become current in the SQL-session that will become the active SQL-session.

4.37.4 Execution contexts

Execution contexts augment an SQL-session context to cater for certain special circumstances that might pertain from time to time during invocations of SQL-statements. An execution context is either a statement execution context, a trigger execution context, or a routine execution context. There is always a *statement execution context*, a *routine execution context*, and zero or more *trigger execution contexts*. For certain SQL-statements, the statement execution context is always *atomic*; for others, it is always or sometimes non-atomic. A routine execution context is either atomic or non-atomic. Every trigger execution context is atomic. Statement execution contexts are described in Subclause 4.33.5, "SQL-statement atomicity and statement execution contexts", routine execution contexts in Subclause 4.37.5, "Routine execution context", and trigger execution contexts in Subclause 4.38.2, "Trigger execution".

4.37.5 Routine execution context

A routine execution context consists of:

— An indication as to whether or not an SQL-invoked routine is active.

— An SQL-data access indication, which identifies what SQL-statements, if any, are allowed during the execution of an SQL-invoked routine. The SQL-data access indication is one of the following: does not possibly contain SQL, possibly contains SQL, possibly reads SQL-data, or possibly modifies SQL-data.

— An identification of the SQL-invoked routine that is active.

— The routine SQL-path derived from the routine SQL-path if the SQL-invoked routine that is active is an SQL routine and from the external routine SQL-path if the SQL-invoked routine that is active is an external routine.

An SQL-invoked routine is active as soon as an SQL-statement executed by an SQL-agent causes invocation of an SQL-invoked routine and ceases to be active when execution of that invocation is complete.

When an SQL-agent causes the invocation of an SQL-invoked routine, a new context for the current SQL-session is created and the values of the current context are preserved. When the execution of that SQL-invoked routine completes, the original context of the current SQL-session is restored and some SQL-session characteristics are reset.

If the routine execution context of the SQL-session indicates that an SQL-invoked routine is active, then the routine SQL-path included in the routine execution context of the SQL-session is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in a <preparable statement> or in a <direct SQL statement>.

4.38 Triggers

4.38.1 General description of triggers

A trigger is a specification for a given action to take place every time a given operation takes place on a given object. The action, known as a *triggered action*, is an SQL-procedure statement or a list of such statements. The object is a persistent base table known as the *subject table* of the trigger. The operation, known as a *trigger event*, is either deletion, insertion, or replacement of a collection of rows.

The triggered action is specified to take place either immediately before the triggering event or immediately after it, according to its specified *trigger action time*, BEFORE or AFTER. The trigger is a *BEFORE trigger* or an *AFTER trigger*, according to its trigger action time.

A trigger is either a *delete trigger*, an *insert trigger*, or an *update trigger*, according to the nature of its trigger event.

Every trigger event arises as a consequence of executing some SQL-data change statement. That consequence might be direct, as for example when the SQL-data change statement is an <insert statement> operating on a base table, or indirect, as for example in the following cases:

— The SQL-data change statement is a <merge statement>.

— The SQL-data change statement operates on the referenced table of some foreign key whose referential action is CASCADE, SET NULL, or SET DEFAULT.

— The SQL-data change statement operates on a viewed table.

A triggered action is permitted to include SQL-data change statements that give rise to trigger events.

A collection of rows being deleted, inserted or replaced is known as a *transition table*. For a delete trigger there is just one transition table, known as an *old transition table*. For an insert trigger there is just one transition table, known as a *new transition table*. For an update trigger there is both an old transition table (the rows being replaced) and a new transition table (the replacement rows), these two tables having the same cardinality.

A reference to "the transition table" of a trigger is ambiguous in the case of an update trigger but whenever such a reference appears in this International Standard it is immaterial to which of the two transition tables it applies.

The triggered action can be specified to take place either just once when the trigger event takes place, in which case the trigger is a *statement-level trigger*, or once for each row of the transition table when the trigger event takes place, in which case the trigger is a *row-level trigger*.

If the triggered action is specified to take place before the event, the trigger is a row-level trigger, and there is a new transition table, then the action can include statements whose effect is to alter the effect of the impending operation.

Special variables make the data in the transition table(s) available to the triggered action. For a statement-level trigger the variable is one whose value is a transition table. For a row-level trigger, the variable is a range variable, known as a *transition variable*. A transition variable ranges over the rows of a transition table, each row giving rise to exactly one execution of the triggered action, with the row in question assigned to the transition

variable. A transition variable is either an *old transition variable* or a *new transition variable*, depending on the transition table over whose rows it ranges.

When there are two transition tables, old and new, each row in the new transition table is one that is derived by an update operation applied to exactly one row in the old transition table. Thus there is a 1:1 correspondence between the rows of the two tables. However, this correspondence is visible only to a row-level trigger, each invocation of which is able to access both the old and new transition variables, the new transition variable representing the result of applying the update operation in question to the row in the old transition variable.

A trigger is defined by a <trigger definition>, specifying the name of the trigger, its subject table, its trigger event, its trigger action time, whether it is statement-level or row-level, names as required for referencing transition tables or variables, and the triggered action.

A schema might include one or more trigger descriptors, each of which includes a triggered action specifying a <triggered SQL statement> that is to be executed (either once for each affected row, in the case of a row-level trigger, or once for the whole trigger event in the case of a statement-level trigger) immediately before or immediately after the trigger event takes place. The execution of a triggered action might cause the triggering of further triggered actions. It does so if it entails execution of an SQL-procedure statement whose effect causes the trigger event of some trigger to take place.

A trigger is described by a trigger descriptor. A trigger descriptor includes:

— The name of the trigger.

— The name of the subject table.

— The trigger action time (BEFORE or AFTER).

— The trigger event (INSERT, DELETE, or UPDATE).

— Whether the trigger is a statement-level trigger or a row-level trigger.

— Any old transition variable name, new transition variable name, old transition table name, or new transition table name.

— The triggered action.

— The trigger column list (possibly empty) for the trigger event.

— The triggered action column set of the triggered action.

— The timestamp of creation of the trigger.

The *order of execution* of a set of triggers is ascending by value of their timestamp of creation in their descriptors, such that the oldest trigger executes first. If one or more triggers have the same timestamp value, then their relative order of execution is implementation-defined.

A triggered action is always executed under the authorization of the owner of the schema that includes the trigger.

4.38.2 Trigger execution

During the execution of an SQL-statement *S*, zero or more *trigger execution contexts* exist, no more than one of which is *active* at any one time. A trigger execution context TEC_i comes into existence, becomes the active one, ceases to be active, and is destroyed as and when required under the General Rules for *S*.

An effect causing a trigger execution context to come into existence here is typically a delete, insert or update operation on one or more base tables, as specified in Subclause 14.16, "Effect of deleting rows from base tables", Subclause 14.19, "Effect of inserting tables into base tables", and Subclause 14.22, "Effect of replacing rows in base tables", respectively.

If, while TEC_i is active, the General Rules for *S* require some new trigger execution context TEC_j to come into existence, then TEC_j replaces TEC_i as the active trigger execution context. TEC_i becomes active again when TEC_j is destroyed.

Multiple trigger execution contexts exist when the General Rules for *S* specify the execution of another SQL-procedure statement *T* before the execution of *S* is complete, and the General Rules for *T* require a new trigger execution context to come into existence.

A trigger execution context consists of a set of *state changes*. Within a trigger execution context, each state change is uniquely identified by a trigger event, a subject table, and a *column list*. The trigger event can be DELETE, INSERT, or UPDATE.

A state change *SC* consists of:

— A *set of transitions*.

— A trigger event.

— A subject table.

— A column list.

— A set (initially empty) of statement-level triggers *considered as executed* for *SC*.

— A set of row-level triggers, each paired with the set of rows in *SC* for which it is considered as executed.

What constitutes a transition depends on the trigger event. If the trigger event is DELETE, a transition is a row in the old transition table. If the trigger event is INSERT, a transition is a row in the new transition table. If the trigger event is UPDATE, a transition is a row *OR* in the old transition table paired with a row *NR* in the new transition table, such that *NR* is the row derived by applying a specified update operation to *OR*. *OR* and *NR* are the *old row* and the *new row*, respectively, of the transition.

A statement-level trigger that is considered as executed for a state change *SC* (in a given trigger execution context) is not subsequently executed for *SC*.

If a row-level trigger *RLT* is considered as executed for some row *R* in *SC*, then *RLT* is not subsequently executed for *R*.

A consequence of the execution of an SQL-data change statement is called an *SQL-update operation* if and only if that consequence causes at least one transition to arise in some state change.

A (possibly empty) old transition table exists if the trigger event is UPDATE or DELETE. It consists of a copy of each row that is to be updated in or deleted from the subject table. A (possibly empty) new transition table

exists if the trigger event is UPDATE or INSERT. It consists of a copy of each row that results from updating a row in the subject table or is to be inserted into the subject table.

A <triggered action> may refer to the old transition table only if an <old transition table name> is specified for it in the <trigger definition>, and to the new transition table only if a <new transition table name> is specified for it in the <trigger definition>.

The <triggered action> of a row-level trigger may refer to a range variable ranging over the rows of the old transition table only if an <old transition variable name> is specified for it in the <trigger definition>. Similarly, the <triggered action> of a row-level trigger may refer to a range variable ranging over the rows of the new transition table only if a <new transition variable name> is specified for it in the <trigger definition>. The scope of a transition variable or transition table name is the <triggered action> of the <trigger definition> that specifies it, excluding any <SQL schema statement>s that are contained in that <triggered action>.

When execution of an SQL-data change statement causes a trigger execution context TEC_i to come into existence, the set of state changes SSC_i in TEC_i is empty. Let $SC_{i,j}$ be a state change in SSC_i. Let *TE* be the trigger event (DELETE, INSERT, or UPDATE) of $SC_{i,j}$. Let *ST* be the subject table of $SC_{i,j}$.

If *TE* is INSERT or DELETE, then let *PSC* be a set whose only element is the empty set.

If *TE* is UPDATE, then:

— Let *CL* be the list of columns being updated by SSC_i.

— Let *OC* be the set of column names identifying the columns in *CL*.

— Let *PSC* be the set consisting of the empty set and every subset of the set of column names of *ST* that has at least one column that is in *OC*.

Let *PSCN* be the number of elements in *PSC*. A state change $SC_{i,j}$, for *j* varying from 1 (one) to *PSCN*, identified by *TE*, *ST*, and the *j*-th element in *PSC*, is added to SSC_i, provided that SSC_i does not already contain a state change corresponding to $SC_{i,j}$. Transitions are added to $SC_{i,j}$ as specified by the General Rules of Subclause 11.8, "<referential constraint definition>", Subclause 14.6, "<delete statement: positioned>", Subclause 14.7, "<delete statement: searched>", Subclause 14.8, "<insert statement>", Subclause 14.10, "<update statement: positioned>", Subclause 14.11, "<update statement: searched>", and Subclause 14.9, "<merge statement>".

When a state change $SC_{i,j}$ arises in SSC_i, one or more triggers are *activated by* $SC_{i,j}$. A trigger *TR* is activated by $SC_{i,j}$ if and only if the subject table of *TR* is the subject table of $SC_{i,j}$, the trigger event of *TR* is the trigger event of $SC_{i,j}$, and the set of column names listed in the trigger column list of *TR* is equivalent to the set of column names listed in $SC_{i,j}$.

NOTE 56 — The trigger column list is included in the descriptor of *TR*; it is empty if the trigger event is DELETE or INSERT. The trigger column list is also empty if the trigger event is UPDATE, but the <trigger event> of the <trigger definition> that defined *TR* does not specify a <trigger column list>.

For each state change $SC_{i,j}$ in TEC_i, the BEFORE triggers activated by $SC_{i,j}$ are executed before any of their triggering events take effect. When those triggering events have taken effect, any AFTER triggers activated by the state changes of TEC_i are executed.

The <triggered action> contained in a <trigger definition> for a BEFORE or AFTER row-level trigger can refer to columns of old transition variables and new transition variables. Such references can be specified as <column

reference>s, which can be <target specification>s and <simple target specification>s when they refer to columns of the new transition variable.

NOTE 57 — By using such <column reference>s as <assignment target>s (see ISO/IEC 9075-4), the triggered action of a BEFORE trigger is able to cause certain SQL-data change statements to have different effects from those specified in the statements.

When an execution of the <triggered SQL statement> *TSS* of a triggered action is not successful, then an exception condition is raised and the SQL-statement that caused *TSS* to be executed has no effect on SQL-data or schemas.

4.39 Client-server operation

When an SQL-agent is active, it is bound in some implementation-defined manner to a single SQL-client. That SQL-client processes the explicit or implicit <SQL connection statement> for the first call to an externally-invoked procedure by an SQL-agent. The SQL-client communicates with, either directly or possibly through other agents such as RDA, one or more SQL-servers. An SQL-session involves an SQL-agent, an SQL-client, and a single SQL-server.

SQL-client modules associated with the SQL-agent exist in the SQL-environment containing the SQL-client associated with the SQL-agent.

Called <externally-invoked procedure>s and <direct SQL statement>s containing an <SQL connection statement> or an <SQL diagnostics statement> are processed by the SQL-client. Following the successful execution of a <connect statement> or a <set connection statement>, the SQL-client modules associated with the SQL-agent are effectively materialized with an implementation-dependent <SQL-client module name> in the SQL-server. Other called <externally-invoked procedure>s and <direct SQL statement>s are processed by the SQL-server.

A call by the SQL-agent to an <externally-invoked procedure> whose <SQL procedure statement> simply contains an <SQL diagnostics statement> fetches information from the specified diagnostics area in the diagnostics area stack associated with the SQL-client. Following the execution of an <SQL procedure statement> by an SQL-server, diagnostic information is passed in an implementation-dependent manner into the SQL-agent's diagnostics area stack in the SQL-client. The effect on diagnostic information of incompatibilities between the character repertoires supported by the SQL-client and SQL-server is implementation-dependent.

This page intentionally left blank.

5 Lexical elements

5.1 <SQL terminal character>

Function

Define the terminal symbols of the SQL language and the elements of strings.

Format

```
<SQL terminal character> ::= <SQL language character>

<SQL language character> ::=
    <simple Latin letter>
  | <digit>
  | <SQL special character>

<simple Latin letter> ::=
    <simple Latin upper case letter>
  | <simple Latin lower case letter>

<simple Latin upper case letter> ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
  | P | Q | R | S | T | U | V | W | X | Y | Z

<simple Latin lower case letter> ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
  | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<SQL special character> ::=
    <space>
  | <double quote>
  | <percent>
  | <ampersand>
  | <quote>
  | <left paren>
  | <right paren>
  | <asterisk>
  | <plus sign>
  | <comma>
  | <minus sign>
  | <period>
  | <solidus>
  | <colon>
  | <semicolon>
```

```
  | <less than operator>
  | <equals operator>
  | <greater than operator>
  | <question mark>
  | <left bracket>
  | <right bracket>
  | <circumflex>
  | <underscore>
  | <vertical bar>
  | <left brace>
  | <right brace>

<space> ::= !! See the Syntax Rules

<double quote> ::= "

<percent> ::= %

<ampersand> ::= &

<quote> ::= '

<left paren> ::= (

<right paren> ::= )

<asterisk> ::= *

<plus sign> ::= +

<comma> ::= ,

<minus sign> ::= -

<period> ::= .

<solidus> ::= /

<reverse solidus> ::= \

<colon> ::= :

<semicolon> ::= ;

<less than operator> ::= <

<equals operator> ::= =

<greater than operator> ::= >

<question mark> ::= ?

<left bracket or trigraph> ::=
    <left bracket>
  | <left bracket trigraph>

<right bracket or trigraph> ::=
    <right bracket>
  | <right bracket trigraph>
```

```
<left bracket> ::= [

<left bracket trigraph> ::= ??(

<right bracket> ::= ]

<right bracket trigraph> ::= ??)

<circumflex> ::= ^

<underscore> ::= _

<vertical bar> ::= |

<left brace> ::= {

<right brace> ::= }
```

Syntax Rules

1) Every character set shall contain a <space> character that is equivalent to U+0020.

Access Rules

None.

General Rules

1) There is a one-to-one correspondence between the symbols contained in <simple Latin upper case letter> and the symbols contained in <simple Latin lower case letter> such that, for all *i*, the symbol defined as the *i*-th alternative for <simple Latin upper case letter> corresponds to the symbol defined as the *i*-th alternative for <simple Latin lower case letter>.

Conformance Rules

None.

5.2 <token> and <separator>

Function

Specify lexical units (tokens and separators) that participate in SQL language.

Format

```
<token> ::=
    <nondelimiter token>
  | <delimiter token>

<nondelimiter token> ::=
    <regular identifier>
  | <key word>
  | <unsigned numeric literal>
  | <national character string literal>
  | <binary string literal>
  | <large object length token>
  | <Unicode delimited identifier>
  | <Unicode character string literal>
  | <SQL language identifier>

<regular identifier> ::= <identifier body>

<identifier body> ::= <identifier start> [ <identifier part>... ]

<identifier part> ::=
    <identifier start>
  | <identifier extend>

<identifier start> ::= !! See the Syntax Rules

<identifier extend> ::= !! See the Syntax Rules

<large object length token> ::= <digit>... <multiplier>

<multiplier> ::=
    K
  | M
  | G

<delimited identifier> ::= <double quote> <delimited identifier body> <double quote>

<delimited identifier body> ::= <delimited identifier part>...

<delimited identifier part> ::=
    <nondoublequote character>
  | <doublequote symbol>

<Unicode delimited identifier> ::=
    U<ampersand><double quote> <Unicode delimiter body> <double quote>
    <Unicode escape specifier>

<Unicode escape specifier> ::= [ UESCAPE <quote><Unicode escape character><quote> ]
```

```
<Unicode delimiter body> ::= <Unicode identifier part>...

<Unicode identifier part> ::=
    <delimited identifier part>
  | <Unicode escape value>

<Unicode escape value> ::=
    <Unicode 4 digit escape value>
  | <Unicode 6 digit escape value>
  | <Unicode character escape value>

<Unicode 4 digit escape value> ::= <Unicode escape character><hexit><hexit><hexit><hexit>

<Unicode 6 digit escape value> ::=
    <Unicode escape character><plus sign>
    <hexit><hexit><hexit><hexit><hexit><hexit>

<Unicode character escape value> ::= <Unicode escape character><Unicode escape character>

<Unicode escape character> ::= !! See the Syntax Rules

<nondoublequote character> ::= !! See the Syntax Rules

<doublequote symbol> ::= "" !! two consecutive double quote characters

<delimiter token> ::=
    <character string literal>
  | <date string>
  | <time string>
  | <timestamp string>
  | <interval string>
  | <delimited identifier>
  | <SQL special character>
  | <not equals operator>
  | <greater than or equals operator>
  | <less than or equals operator>
  | <concatenation operator>
  | <right arrow>
  | <left bracket trigraph>
  | <right bracket trigraph>
  | <double colon>
  | <double period>

<not equals operator> ::= <>

<greater than or equals operator> ::= >=

<less than or equals operator> ::= <=

<concatenation operator> ::= ||

<right arrow> ::= ->

<double colon> ::= ::

<double period> ::= ..

<separator> ::= { <comment> | <white space> }...
```

```
<white space> ::= !! See the Syntax Rules

<comment> ::=
    <simple comment>
  | <bracketed comment>

<simple comment> ::= <simple comment introducer> [ <comment character>... ] <newline>

<simple comment introducer> ::= <minus sign><minus sign>

<bracketed comment> ::=
    <bracketed comment introducer>
    <bracketed comment contents>
    <bracketed comment terminator>

<bracketed comment introducer> ::= /*

<bracketed comment terminator> ::= */

<bracketed comment contents> ::= !! See the Syntax Rules
    [ { <comment character> | <separator> }... ]

<comment character> ::=
    <nonquote character>
  | <quote>

<newline> ::= !! See the Syntax Rules

<key word> ::=
    <reserved word>
  | <non-reserved word>

<non-reserved word> ::=
    A | ABSOLUTE | ACTION | ADA | ADD | ADMIN | AFTER | ALWAYS | ASC
  | ASSERTION | ASSIGNMENT | ATTRIBUTE | ATTRIBUTES

  | BEFORE | BERNOULLI | BREADTH

  | C | CASCADE | CATALOG | CATALOG_NAME | CHAIN | CHARACTER_SET_CATALOG
  | CHARACTER_SET_NAME | CHARACTER_SET_SCHEMA | CHARACTERISTICS | CHARACTERS
  | CLASS_ORIGIN | COBOL | COLLATION | COLLATION_CATALOG | COLLATION_NAME | COLLATION_SCHEMA
  | COLUMN_NAME | COMMAND_FUNCTION | COMMAND_FUNCTION_CODE | COMMITTED
  | CONDITION_NUMBER | CONNECTION | CONNECTION_NAME | CONSTRAINT_CATALOG | CONSTRAINT_NAME
  | CONSTRAINT_SCHEMA | CONSTRAINTS | CONSTRUCTOR | CONTAINS | CONTINUE | CURSOR_NAME

  | DATA | DATETIME_INTERVAL_CODE | DATETIME_INTERVAL_PRECISION | DEFAULTS | DEFERRABLE
  | DEFERRED | DEFINED | DEFINER | DEGREE | DEPTH | DERIVED | DESC | DESCRIPTOR
  | DIAGNOSTICS | DISPATCH | DOMAIN | DYNAMIC_FUNCTION | DYNAMIC_FUNCTION_CODE

  | EQUALS | EXCEPTION | EXCLUDE | EXCLUDING

  | FINAL | FIRST | FOLLOWING | FORTRAN | FOUND

  | G | GENERAL | GENERATED | GO | GOTO | GRANTED

  | HIERARCHY
```

```
  | IMMEDIATE | IMPLEMENTATION | INCLUDING | INCREMENT | INITIALLY | INPUT | INSTANCE
  | INSTANTIABLE | INVOKER | ISOLATION

  | K | KEY | KEY_MEMBER | KEY_TYPE

  | LAST | LENGTH | LEVEL | LOCATOR

  | M | MAP | MATCHED | MAXVALUE | MESSAGE_LENGTH | MESSAGE_OCTET_LENGTH
  | MESSAGE_TEXT | MINVALUE | MORE | MUMPS

  | NAME | NAMES | NESTING | NEXT | NORMALIZED | NULLABLE | NULLS | NUMBER

  | OBJECT | OCTETS | OPTION | OPTIONS | ORDERING | ORDINALITY | OTHERS
  | OUTPUT | OVERRIDING

  | PAD | PARAMETER_MODE | PARAMETER_NAME | PARAMETER_ORDINAL_POSITION
  | PARAMETER_SPECIFIC_CATALOG | PARAMETER_SPECIFIC_NAME | PARAMETER_SPECIFIC_SCHEMA
  | PARTIAL | PASCAL | PATH | PLACING | PLI | PRECEDING | PRESERVE | PRIOR
  | PRIVILEGES | PUBLIC

  | READ | RELATIVE | REPEATABLE | RESTART | RESTRICT | RETURNED_CARDINALITY
  | RETURNED_LENGTH | RETURNED_OCTET_LENGTH | RETURNED_SQLSTATE | ROLE
  | ROUTINE | ROUTINE_CATALOG | ROUTINE_NAME | ROUTINE_SCHEMA | ROW_COUNT

  | SCALE | SCHEMA | SCHEMA_NAME | SCOPE_CATALOG | SCOPE_NAME | SCOPE_SCHEMA
  | SECTION | SECURITY | SELF | SEQUENCE | SERIALIZABLE | SERVER_NAME | SESSION
  | SETS | SIMPLE | SIZE | SOURCE | SPACE | SPECIFIC_NAME | STATE | STATEMENT
  | STRUCTURE | STYLE | SUBCLASS_ORIGIN

  | TABLE_NAME | TEMPORARY | TIES | TOP_LEVEL_COUNT | TRANSACTION
  | TRANSACTION_ACTIVE | TRANSACTIONS_COMMITTED | TRANSACTIONS_ROLLED_BACK
  | TRANSFORM | TRANSFORMS | TRIGGER_CATALOG | TRIGGER_NAME | TRIGGER_SCHEMA | TYPE

  | UNBOUNDED | UNCOMMITTED | UNDER | UNNAMED | USAGE | USER_DEFINED_TYPE_CATALOG
  | USER_DEFINED_TYPE_CODE | USER_DEFINED_TYPE_NAME | USER_DEFINED_TYPE_SCHEMA

  | VIEW

  | WORK | WRITE

  | ZONE

<reserved word> ::=
    ABS | ALL | ALLOCATE | ALTER | AND | ANY | ARE | ARRAY | AS | ASENSITIVE
  | ASYMMETRIC | AT | ATOMIC | AUTHORIZATION | AVG

  | BEGIN | BETWEEN | BIGINT | BINARY | BLOB | BOOLEAN | BOTH | BY

  | CALL | CALLED | CARDINALITY | CASCADED | CASE | CAST | CEIL | CEILING
  | CHAR | CHAR_LENGTH | CHARACTER | CHARACTER_LENGTH | CHECK | CLOB | CLOSE
  | COALESCE | COLLATE | COLLECT | COLUMN | COMMIT | CONDITION | CONNECT
  | CONSTRAINT | CONVERT | CORR | CORRESPONDING | COUNT | COVAR_POP | COVAR_SAMP
  | CREATE | CROSS | CUBE | CUME_DIST | CURRENT | CURRENT_DATE
  | CURRENT_DEFAULT_TRANSFORM_GROUP | CURRENT_PATH | CURRENT_ROLE | CURRENT_TIME
  | CURRENT_TIMESTAMP | CURRENT_TRANSFORM_GROUP_FOR_TYPE | CURRENT_USER
  | CURSOR | CYCLE
```

```
| DATE | DAY | DEALLOCATE | DEC | DECIMAL | DECLARE | DEFAULT | DELETE
| DENSE_RANK | DEREF | DESCRIBE | DETERMINISTIC | DISCONNECT | DISTINCT
| DOUBLE | DROP | DYNAMIC

| EACH | ELEMENT | ELSE | END | END-EXEC | ESCAPE | EVERY | EXCEPT | EXEC
| EXECUTE | EXISTS | EXP | EXTERNAL | EXTRACT

| FALSE | FETCH | FILTER | FLOAT | FLOOR | FOR | FOREIGN | FREE | FROM
| FULL | FUNCTION | FUSION

| GET | GLOBAL | GRANT | GROUP | GROUPING

| HAVING | HOLD | HOUR

| IDENTITY | IN | INDICATOR | INNER | INOUT | INSENSITIVE | INSERT
| INT | INTEGER | INTERSECT | INTERSECTION | INTERVAL | INTO | IS

| JOIN

| LANGUAGE | LARGE | LATERAL | LEADING | LEFT | LIKE | LN | LOCAL
| LOCALTIME | LOCALTIMESTAMP | LOWER

| MATCH | MAX | MEMBER | MERGE | METHOD | MIN | MINUTE
| MOD | MODIFIES | MODULE | MONTH | MULTISET

| NATIONAL | NATURAL | NCHAR | NCLOB | NEW | NO | NONE | NORMALIZE | NOT
| NULL | NULLIF | NUMERIC

| OCTET_LENGTH | OF | OLD | ON | ONLY | OPEN | OR | ORDER | OUT | OUTER
| OVER | OVERLAPS | OVERLAY

| PARAMETER | PARTITION | PERCENT_RANK | PERCENTILE_CONT | PERCENTILE_DISC
| POSITION | POWER | PRECISION | PREPARE | PRIMARY | PROCEDURE

| RANGE | RANK | READS | REAL | RECURSIVE | REF | REFERENCES | REFERENCING
| REGR_AVGX | REGR_AVGY | REGR_COUNT | REGR_INTERCEPT | REGR_R2 | REGR_SLOPE
| REGR_SXX | REGR_SXY | REGR_SYY | RELEASE | RESULT | RETURN | RETURNS
| REVOKE | RIGHT | ROLLBACK | ROLLUP | ROW | ROW_NUMBER | ROWS

| SAVEPOINT | SCOPE | SCROLL | SEARCH | SECOND | SELECT | SENSITIVE
| SESSION_USER | SET | SIMILAR | SMALLINT | SOME | SPECIFIC | SPECIFICTYPE
| SQL | SQLEXCEPTION | SQLSTATE | SQLWARNING | SQRT | START | STATIC
| STDDEV_POP | STDDEV_SAMP | SUBMULTISET | SUBSTRING | SUM | SYMMETRIC
| SYSTEM | SYSTEM_USER

| TABLE | TABLESAMPLE | THEN | TIME | TIMESTAMP | TIMEZONE_HOUR | TIMEZONE_MINUTE
| TO | TRAILING | TRANSLATE | TRANSLATION | TREAT | TRIGGER | TRIM | TRUE

| UESCAPE | UNION | UNIQUE | UNKNOWN | UNNEST | UPDATE  | UPPER | USER | USING

| VALUE | VALUES | VAR_POP | VAR_SAMP | VARCHAR | VARYING

| WHEN | WHENEVER | WHERE | WIDTH_BUCKET | WINDOW | WITH | WITHIN | WITHOUT

| YEAR
```

Syntax Rules

1) An <identifier start> is any character in the Unicode General Category classes "Lu", "Ll", "Lt", "Lm", "Lo", or "Nl".

   NOTE 58 — The Unicode General Category classes "Lu", "Ll", "Lt", "Lm", "Lo", and "Nl" are assigned to Unicode characters that are, respectively, upper-case letters, lower-case letters, title-case letters, modifier letters, other letters, and letter numbers.

2) An <identifier extend> is U+00B7, "Middle Dot", or any character in the Unicode General Category classes "Mn", "Mc", "Nd", "Pc", or "Cf".

   NOTE 59 — The Unicode General Category classes "Mn", "Mc", "Nd", "Pc", and "Cf" are assigned to Unicode characters that are, respectively, nonspacing marks, spacing combining marks, decimal numbers, connector punctuations, and formatting codes.

3) <white space> is any consecutive sequence of characters each of which satisfies the definition of white space found in Subclause 3.1.6, "Definitions provided in Part 2".

4) <newline> is the implementation-defined end-of-line indicator.

   NOTE 60 — <newline> is typically represented by U+000A ("Line Feed") and/or U+000D ("Carriage Return"); however, this representation is not required by ISO/IEC 9075.

5) With the exception of the <space> character explicitly contained in <timestamp string> and <interval string>, a <token>, other than a <character string literal>, a <national character string literal>, a <Unicode character string literal>, a <delimited identifier>, or a <Unicode delimited identifier> shall not contain a <space> character or other <separator>.

6) A <nondoublequote character> is any character of the source language character set other than a <double quote>.

   NOTE 61 — "source language character set" is defined in Subclause 4.8.1, "Host languages", in ISO/IEC 9075-1.

7) Any <token> may be followed by a <separator>. A <nondelimiter token> shall be followed by a <delimiter token> or a <separator>.

   NOTE 62 — If the Format does not allow a <nondelimiter token> to be followed by a <delimiter token>, then that <nondelimiter token> shall be followed by a <separator>.

8) There shall be no <separator> separating the <minus sign>s of a <simple comment introducer>.

9) There shall be no <separator> separating any two <digit>s or separating a <digit> and <multiplier> of a <large object length token>.

10) Within a <bracketed comment contents>, any <solidus> immediately followed by an <asterisk> without any intervening <separator> shall be considered to be the <bracketed comment introducer> of a <separator> that is a <bracketed comment>.

    NOTE 63 — Conforming programs should not place <simple comment> within a <bracketed comment> because if such a <simple comment> contains the sequence of characters "*/" without a preceding "/*" in the same <simple comment>, it will prematurely terminate the containing <bracketed comment>.

11) SQL text containing one or more instances of <comment> is equivalent to the same SQL text with the <comment> replaced with <newline>.

12) In a <regular identifier>, the number of <identifier part>s shall be less than 128.

13) The <delimited identifier body> of a <delimited identifier> shall not comprise more than 128 <delimited identifier part>s.

14) In a <Unicode delimited identifier>, there shall be no <separator> between the 'U' and the <ampersand> nor between the <ampersand> and the <double quote>.

15) <Unicode escape character> shall be a single character from the source language character set other than a <hexit>, <plus sign>, <double quote>, or <white space>.

16) If the source language character set contains <reverse solidus>, then let *DEC* be <reverse solidus>; otherwise, let *DEC* be an implementation-defined character from the source language character set that is not a <hexit>, <plus sign>, <double quote>, or <white space>.

17) If a <Unicode escape specifier> does not contain <Unicode escape character>, then "UESCAPE <quote>*DEC*<quote>" is implicit.

18) In a <Unicode escape value> there shall be no <separator> between the <Unicode escape character> and the first <hexit>, nor between any of the <hexit>s.

19) The <Unicode delimiter body> of a <Unicode delimited identifier> shall not comprise more than 128 <Unicode identifier part>s.

20) <Unicode 4 digit escape value> '<Unicode escape character>xyzw' is equivalent to the Unicode code point specified by U+xyzw.

21) <Unicode 6 digit escape value> '<Unicode escape character>+xyzwrs' is equivalent to the character at the Unicode code point specified by U+xyzwrs.

NOTE 64 — The 6-hexit notation is derived by taking the UCS-4 notation defined in ISO/IEC 10646-1 and removing the leading two hexits, whose values are always 0 (zero).

22) <Unicode character escape value> is equivalent to a single instance of <Unicode escape character>.

23) For every <identifier body> *IB* there is exactly one corresponding case-normal form *CNF*. *CNF* is an <identifier body> derived from *IB* as follows.

Let n be the number of characters in *IB*. For i ranging from 1 (one) to n, the i-th character M_i of *IB* is transliterated into the corresponding character or characters of *CNF* as follows.

Case:

a) If M_i is a lower case character or a title case character for which an equivalent upper case sequence U is defined by Unicode, then let j be the number of characters in U; the next j characters of *CNF* are U.

b) Otherwise, the next character of *CNF* is M_i.

24) The case-normal form of the <identifier body> of a <regular identifier> is used for purposes such as and including determination of identifier equivalence, representation in the Definition and Information Schemas, and representation in diagnostics areas.

NOTE 65 — The Information Schema and Definition Schema are defined in ISO/IEC 9075-11.

NOTE 66 — Any lower-case letters for which there are no upper-case equivalents are left in their lower-case form.

25) The case-normal form of <regular identifier> shall not be equal, according to the comparison rules in Subclause 8.2, "<comparison predicate>", to any <reserved word> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER.

26) Two <regular identifier>s are equivalent if the case-normal forms of their <identifier body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation *IDC* that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

27) A <regular identifier> and a <delimited identifier> are equivalent if the case-normal form of the <identifier body> of the <regular identifier> and the <delimited identifier body> of the <delimited identifier> (with all occurrences of <quote> replaced by <quote symbol> and all occurrences of <doublequote symbol> replaced by <double quote>), considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and *IDC*, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

28) Two <delimited identifier>s are equivalent if their <delimited identifier body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

29) Two <Unicode delimited identifier>s are equivalent if their <Unicode delimiter body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

30) A <Unicode delimited identifier> and a <delimited identifier> are equivalent if their <Unicode delimiter body> and <delimited identifier body>, respectively, each considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

31) For the purposes of identifying <key word>s, any <simple Latin lower case letter> contained in a candidate <key word> shall be effectively treated as the corresponding <simple Latin upper case letter>.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F391, "Long identifiers", in a <regular identifier>, the number of <identifier part>s shall be less than 18.

2) Without Feature F391, "Long identifiers", the <delimited identifier body> of a <delimited identifier> shall not comprise more than 18 <delimited identifier part>s.

NOTE 67 — Not every character set supported by a conforming SQL-implementation necessarily contains every character associated with <identifier start> and <identifier part> that is identified in the Syntax Rules of this Subclause. No conforming SQL-implementation shall be required to support in <identifier start> or <identifier part> any character identified in the Syntax Rules of this Subclause unless that character belongs to the character set in use for an SQL-client module or in SQL-data.

3) Without Feature T351, "Bracketed comments", conforming SQL language shall not contain a <bracketed comment>.

4) Without Feature F392, "Unicode escapes in identifiers", conforming SQL language shall not contain a <Unicode delimited identifier>.

5.3 <literal>

Function

Specify a non-null value.

Format

```
<literal> ::=
    <signed numeric literal>
  | <general literal>

<unsigned literal> ::=
    <unsigned numeric literal>
  | <general literal>

<general literal> ::=
    <character string literal>
  | <national character string literal>
  | <Unicode character string literal>
  | <binary string literal>
  | <datetime literal>
  | <interval literal>
  | <boolean literal>

<character string literal> ::=
    [ <introducer><character set specification> ]
    <quote> [ <character representation>... ] <quote>
    [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<introducer> ::= <underscore>

<character representation> ::=
    <nonquote character>
  | <quote symbol>

<nonquote character> ::= !! See the Syntax Rules.

<quote symbol> ::= <quote><quote>

<national character string literal> ::=
    N <quote> [ <character representation>... ]
    <quote> [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<Unicode character string literal> ::=
    [ <introducer><character set specification> ]
    U<ampersand><quote> [ <Unicode representation>... ] <quote>
    [ { <separator> <quote> [ <Unicode representation>... ] <quote> }... ]
    <Unicode escape specifier>

<Unicode representation> ::=
    <character representation>
  | <Unicode escape value>
```

```
<binary string literal> ::=
    X <quote> [ { <hexit> <hexit> }... ] <quote>
    [ { <separator> <quote> [ { <hexit> <hexit> }... ] <quote> }... ]

<hexit> ::=
    <digit> | A | B | C | D | E | F | a | b | c | d | e | f

<signed numeric literal> ::= [ <sign> ] <unsigned numeric literal>

<unsigned numeric literal> ::=
    <exact numeric literal>
  | <approximate numeric literal>

<exact numeric literal> ::=
    <unsigned integer> [ <period> [ <unsigned integer> ] ]
  | <period> <unsigned integer>

<sign> ::=
    <plus sign>
  | <minus sign>

<approximate numeric literal> ::= <mantissa> E <exponent>

<mantissa> ::= <exact numeric literal>

<exponent> ::= <signed integer>

<signed integer> ::= [ <sign> ] <unsigned integer>

<unsigned integer> ::= <digit>...

<datetime literal> ::=
    <date literal>
  | <time literal>
  | <timestamp literal>

<date literal> ::= DATE <date string>

<time literal> ::= TIME <time string>

<timestamp literal> ::= TIMESTAMP <timestamp string>

<date string> ::= <quote> <unquoted date string> <quote>

<time string> ::= <quote> <unquoted time string> <quote>

<timestamp string> ::= <quote> <unquoted timestamp string> <quote>

<time zone interval> ::= <sign> <hours value> <colon> <minutes value>

<date value> ::= <years value> <minus sign> <months value> <minus sign> <days value>

<time value> ::= <hours value> <colon> <minutes value> <colon> <seconds value>

<interval literal> ::= INTERVAL [ <sign> ] <interval string> <interval qualifier>

<interval string> ::= <quote> <unquoted interval string> <quote>

<unquoted date string> ::= <date value>
```

```
<unquoted time string> ::= <time value> [ <time zone interval> ]

<unquoted timestamp string> ::= <unquoted date string> <space> <unquoted time string>

<unquoted interval string> ::=
    [ <sign> ] { <year-month literal> | <day-time literal> }

<year-month literal> ::=
    <years value> [ <minus sign> <months value> ]
  | <months value>

<day-time literal> ::=
    <day-time interval>
  | <time interval>

<day-time interval> ::=
    <days value> [ <space> <hours value> [ <colon> <minutes value>
    [ <colon> <seconds value> ] ] ]

<time interval> ::=
    <hours value> [ <colon> <minutes value> [ <colon> <seconds value> ] ]
  | <minutes value> [ <colon> <seconds value> ]
  | <seconds value>

<years value> ::= <datetime value>

<months value> ::= <datetime value>

<days value> ::= <datetime value>

<hours value> ::= <datetime value>

<minutes value> ::= <datetime value>

<seconds value> ::= <seconds integer value> [ <period> [ <seconds fraction> ] ]

<seconds integer value> ::= <unsigned integer>

<seconds fraction> ::= <unsigned integer>

<datetime value> ::= <unsigned integer>

<boolean literal> ::=
    TRUE
  | FALSE
  | UNKNOWN
```

Syntax Rules

1) In a <character string literal> or <national character string literal>, the sequence:

   <quote> <character representation>... <quote> <separator> <quote> <character representation>... <quote>

   is equivalent to the sequence

   <quote> <character representation>... <character representation>... <quote>

NOTE 68 — The <character representation>s in the equivalent sequence are in the same sequence and relative sequence as in the original <character string literal>.

2) In a <Unicode character string literal>, the sequence:

<quote> <Unicode representation>... <quote> <separator> <quote> <Unicode representation>... <quote>

is equivalent to the sequence:

<quote> <Unicode representation>... <Unicode representation>... <quote>

3) In a <binary string literal>, the sequence

<quote> { <hexit> <hexit> }... <quote> <separator> <quote> { <hexit> <hexit> }... <quote>

is equivalent to the sequence

<quote> { <hexit> <hexit> }... { <hexit> <hexit> }... <quote>

NOTE 69 — The <hexit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <binary string literal>.

4) In a <character string literal>, <national character string literal>, <Unicode character string literal>, or <binary string literal>, a <separator> shall contain a <newline>.

5) A <national character string literal> is equivalent to a <character string literal> with the "N" replaced by "<introducer><character set specification>", where "<character set specification>" is an implementation-defined <character set name>.

6) In a <Unicode character string literal> that specifies "<introducer><character set specification>", there shall be no <separator> between the <introducer> and the <character set specification>.

7) In a <Unicode character string literal>, there shall be no <separator> between the "U" and the <ampersand> nor between the <ampersand> and the <quote>.

8) The character set of a <Unicode character string literal> that specifies "<introducer><character set specification>" is the character set specified by the <character set specification>. The character set of a <Unicode character string literal> that does not specify "<introducer><character set specification>" is the character set of the SQL-client module that contains the <Unicode character string literal>.

9) A <Unicode character string literal> is equivalent to a <character string literal> in which every <Unicode escape value> has been replaced with the equivalent Unicode character. The set of characters contained in the <Unicode character string literal> shall be wholly contained in the character set of the <Unicode character string literal>.

NOTE 70 — The requirement for "wholly contained" applies after the replacement of <Unicode escape value>s with equivalent Unicode characters.

10) Each <character representation> is a character of the source language character set. The value of a <character string literal>, viewed as a string in the source language character set, shall be equivalent to a character string of the implicit or explicit character set of the <character string literal> or <national character string literal>.

NOTE 71 — "source language character set" is defined in Subclause 4.8.1, "Host languages", in ISO/IEC 9075-1.

11) A <nonquote character> is one of:

a) Any character of the source language character set other than a <quote>.

b) Any character other than a <quote> in the character set identified by the <character set specification> or implied by "N".

12) Case:

a) If a <character set specification> is not specified in a <character string literal>, then the set of characters contained in the <character string literal> shall be wholly contained in the character set of the <SQL-client module definition> that contains the <character string literal>.

b) Otherwise, there shall be no <separator> between the <introducer> and the <character set specification>, and the set of characters contained in the <character string literal> shall be wholly contained in the character set specified by the <character set specification>.

13) The declared type of a <character string literal> is fixed-length character string. The length of a <character string literal> is the number of <character representation>s that it contains. Each <quote symbol> contained in <character string literal> represents a single <quote> in both the value and the length of the <character string literal>. The two <quote>s contained in a <quote symbol> shall not be separated by any <separator>.

NOTE 72 — <character string literal>s are allowed to be zero-length strings (*i.e.*, to contain no characters) even though it is not permitted to declare a <data type> that is CHARACTER with <length> 0 (zero).

14) The character set of a <character string literal> is

Case:

a) If the <character string literal> specifies a <character set specification>, then the character set specified by that <character set specification>.

b) Otherwise, the character set of the SQL-client module that contains the <character string literal>.

15) The declared type collation of a <character string literal> is the character set collation, and the collation derivation is *implicit*.

16) The declared type of a <binary string literal> is binary string. Each <hexit> appearing in the literal is equivalent to a quartet of bits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are interpreted as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111, respectively. The <hexit>s a, b, c, d, e, and f have respectively the same values as the <hexit>s A, B, C, D, E, and F.

17) An <exact numeric literal> without a <period> has an implied <period> following the last <digit>.

18) The declared type of an <exact numeric literal> *ENL* is an implementation-defined exact numeric type whose scale is the number of <digit>s to the right of the <period>. There shall be an exact numeric type capable of representing the value of *ENL* exactly.

19) The declared type of an <approximate numeric literal> *ANL* is an implementation-defined approximate numeric type. The value of *ANL* shall not be greater than the maximum value nor less than the minimum value that can be represented by the approximate numeric types.

NOTE 73 — Thus the only syntax error for an <approximate numeric literal> is what is commonly known as "overflow"; there is no syntax error for specifying more significant digits than the SQL-implementation can represent internally, nor for specifying a value that has no exact equivalent in the SQL-implementation's internal representation. ("Underflow", *i.e.*, specifying a nonzero value so close to 0 (zero) that the closest representation in the SQL-implementation's internal representation is 0E0, is a special case of the latter condition, and is not a syntax error.)

20) The declared type of a <date literal> is DATE.

21) The declared type of a <time literal> that does not specify <time zone interval> is TIME(*P*) WITHOUT TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise. The declared type of a <time literal> that specifies <time zone interval> is TIME(*P*) WITH TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise.

22) The declared type of a <timestamp literal> that does not specify <time zone interval> is TIMESTAMP(*P*) WITHOUT TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise. The declared type of a <timestamp literal> that specifies <time zone interval> is TIMESTAMP(*P*) WITH TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise.

23) If <time zone interval> is not specified, then the effective <time zone interval> of the datetime data type is the current default time zone displacement for the SQL-session.

24) Let *datetime component* be either <years value>, <months value>, <days value>, <hours value>, <minutes value>, or <seconds value>.

25) Let *N* be the number of <primary datetime field>s in the precision of the <interval literal>, as specified by <interval qualifier>.

    The <interval literal> being defined shall contain *N* datetime components.

    The declared type of <interval literal> specified with an <interval qualifier> is INTERVAL with the <interval qualifier>.

    Each datetime component shall have the precision specified by the <interval qualifier>.

26) Within a <datetime literal>, the <years value> shall contain four digits. The <seconds integer value> and other datetime components, with the exception of <seconds fraction>, shall each contain two digits.

27) Within the definition of a <datetime literal>, the <datetime value>s are constrained by the natural rules for dates and times according to the Gregorian calendar.

28) Within the definition of an <interval literal>, the <datetime value>s are constrained by the natural rules for intervals according to the Gregorian calendar.

29) Within the definition of an <interval literal> that contains a <year-month literal>, the <interval qualifier> shall not specify DAY, HOUR, MINUTE, or SECOND. Within the definition of an <interval literal> that contains a <day-time literal>, the <interval qualifier> shall not specify YEAR or MONTH.

30) Within the definition of a <datetime literal>, the value of the <time zone interval> shall be in the range –12:59 to +14:00.

Access Rules

None.

General Rules

1) The value of a <character string literal> is the result of transliterating the sequence of <character representation>s that it contains from the source language character set to the implicit or explicit character set of the <character string literal>.

2) If the character repertoire of a <character string literal> *US* is UCS, then its value is replaced by NORMALIZE(*US*).

3) The numeric value of an <exact numeric literal> is determined by the normal mathematical interpretation of positional decimal notation.

4) The numeric value of an <approximate numeric literal> is approximately the product of the exact numeric value represented by the <mantissa> with the number obtained by raising the number 10 to the power of the exact numeric value represented by the <exponent>.

5) The <sign> in a <signed numeric literal> or an <interval literal> is a monadic arithmetic operator. The monadic arithmetic operators + and – specify monadic plus and monadic minus, respectively. If neither monadic plus nor monadic minus are specified in a <signed numeric literal> or an <interval literal>, or if monadic plus is specified, then the literal is positive. If monadic minus is specified in a <signed numeric literal> or <interval literal>, then the literal is negative. If <sign> is specified in both possible locations in an <interval literal>, then the sign of the literal is determined by normal mathematical interpretation of multiple sign operators.

6) Let *V* be the integer value of the <unsigned integer> contained in <seconds fraction> and let *N* be the number of digits in the <seconds fraction> respectively. The resultant value of the <seconds fraction> is effectively determined as follows:

   Case:

   a) If <seconds fraction> is specified within the definition of a <datetime literal>, then the effective value of the <seconds fraction> is $V * 10^{-N}$ seconds.

   b) If <seconds fraction> is specified within the definition of an <interval literal>, then let *M* be the <interval fractional seconds precision> specified in the <interval qualifier>.

      Case:

      i) If $N < M$, then let *V1* be $V * 10^{M-N}$; the effective value of the <seconds fraction> is $V1 * 10^{-M}$ seconds.

      ii) If $N > M$, then let *V2* be the integer part of the quotient of $V/10^{N-M}$; the effective value of the <seconds fraction> is $V2 * 10^{-M}$ seconds.

      iii) Otherwise, the effective value of the <seconds fraction> is $V * 10^{-M}$ seconds.

7) The *i*-th datetime component in a <datetime literal> or <interval literal> assigns the value of the datetime component to the *i*-th <primary datetime field> in the <datetime literal> or <interval literal>.

8) If <time zone interval> is specified, then the time and timestamp values in <time literal> and <timestamp literal> represent a datetime in the specified time zone.

9) If <date value> is specified, then it is interpreted as a date in the Gregorian calendar. If <time value> is specified, then it is interpreted as a time of day. Let *DV* be the value of the <datetime literal>, disregarding <time zone interval>.

   Case:

a) If <time zone interval> is specified, then let *TZI* be the value of the interval denoted by <time zone interval>. The value of the <datetime literal> is *DV* – *TZI*, with time zone displacement *TZI*.

b) Otherwise, the value of the <datetime literal> is *DV*.

NOTE 74 — If <time zone interval> is specified, then a <time literal> or <timestamp literal> is interpreted as local time with the specified time zone displacement. However, it is effectively converted to UTC while retaining the original time zone displacement.

If <time zone interval> is not specified, then no assumption is made about time zone displacement. However, should a time zone displacement be required during subsequent processing, the current default time zone displacement of the SQL-session will be applied at that time.

10) The truth value of a <boolean literal> is *True* if TRUE is specified, is *False* if FALSE is specified, and is *Unknown* if UNKNOWN is specified.

NOTE 75 — The null value of the boolean data type is equivalent to the *Unknown* truth value (see Subclause 4.5, "Boolean types").

Conformance Rules

1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean literal>.

2) Without Feature F555, "Enhanced seconds precision", in conforming SQL language, an <unsigned integer> that is a <seconds fraction> that is contained in a <timestamp literal> shall not contain more than 6 <digit>s.

3) Without Feature F555, "Enhanced seconds precision", in conforming SQL language, a <time literal> shall not contain a <seconds fraction>.

4) Without Feature F421, "National character", conforming SQL language shall not contain a <national character string literal>.

5) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval literal>.

6) Without Feature F271, "Compound character literals", in conforming SQL language, a <character string literal> shall contain exactly one repetition of <character representation> (that is, it shall contain exactly one sequence of "<quote> <character representation>... <quote>").

7) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <time zone interval>.

8) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <binary string literal>.

9) Without Feature F393, "Unicode escapes in literals", conforming SQL language shall not contain a <Unicode character string literal>.

5.4 Names and identifiers

Function

Specify names.

Format

```
<identifier> ::= <actual identifier>

<actual identifier> ::=
    <regular identifier>
  | <delimited identifier>
  | <Unicode delimited identifier>

<SQL language identifier> ::=
    <SQL language identifier start> [ <SQL language identifier part>... ]

<SQL language identifier start> ::= <simple Latin letter>

<SQL language identifier part> ::=
    <simple Latin letter>
  | <digit>
  | <underscore>

<authorization identifier> ::=
    <role name>
  | <user identifier>

<table name> ::= <local or schema qualified name>

<domain name> ::= <schema qualified name>

<schema name> ::= [ <catalog name> <period> ] <unqualified schema name>

<unqualified schema name> ::= <identifier>

<catalog name> ::= <identifier>

<schema qualified name> ::= [ <schema name> <period> ] <qualified identifier>

<local or schema qualified name> ::=
    [ <local or schema qualifier> <period> ] <qualified identifier>

<local or schema qualifier> ::=
    <schema name>
  | <local qualifier>

<qualified identifier> ::= <identifier>

<column name> ::= <identifier>

<correlation name> ::= <identifier>

<query name> ::= <identifier>
```

```
<SQL-client module name> ::= <identifier>

<procedure name> ::= <identifier>

<schema qualified routine name> ::= <schema qualified name>

<method name> ::= <identifier>

<specific name> ::= <schema qualified name>

<cursor name> ::= <local qualified name>

<local qualified name> ::= [ <local qualifier> <period> ] <qualified identifier>

<local qualifier> ::= MODULE

<host parameter name> ::= <colon> <identifier>

<SQL parameter name> ::= <identifier>

<constraint name> ::= <schema qualified name>

<external routine name> ::=
    <identifier>
  | <character string literal>

<trigger name> ::= <schema qualified name>

<collation name> ::= <schema qualified name>

<character set name> ::= [ <schema name> <period> ] <SQL language identifier>

<transliteration name> ::= <schema qualified name>

<transcoding name> ::= <schema qualified name>

<schema-resolved user-defined type name> ::= <user-defined type name>

<user-defined type name> ::= [ <schema name> <period> ] <qualified identifier>

<attribute name> ::= <identifier>

<field name> ::= <identifier>

<savepoint name> ::= <identifier>

<sequence generator name> ::= <schema qualified name>

<role name> ::= <identifier>

<user identifier> ::= <identifier>

<connection name> ::= <simple value specification>

<SQL-server name> ::= <simple value specification>

<connection user name> ::= <simple value specification>

<SQL statement name> ::=
    <statement name>
```

```
  | <extended statement name>

<statement name> ::= <identifier>

<extended statement name> ::= [ <scope option> ] <simple value specification>

<dynamic cursor name> ::=
    <cursor name>
  | <extended cursor name>

<extended cursor name> ::= [ <scope option> ] <simple value specification>

<descriptor name> ::= [ <scope option> ] <simple value specification>

<scope option> ::=
    GLOBAL
  | LOCAL

<window name> ::= <identifier>
```

Syntax Rules

1) In an <SQL language identifier>, the number of <SQL language identifier part>s shall be less than 128.

2) An <SQL language identifier> is equivalent to an <SQL language identifier> in which every letter that is a lower-case letter is replaced by the corresponding upper-case letter or letters. This treatment includes determination of equivalence, representation in the Information and Definition Schemas, representation in diagnostics areas, and similar uses.

   NOTE 76 — The Information Schema and Definition Schema are defined in ISO/IEC 9075-11.

3) An <SQL language identifier> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER, shall not be equal, according to the comparison rules in Subclause 8.2, "<comparison predicate>", to any <reserved word> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER.

   NOTE 77 — It is the intention that no <key word> specified in ISO/IEC 9075 or revisions thereto shall end with an <underscore>.

4) If a <local or schema qualified name> does not contain a <local or schema qualifier>, then

   Case:

   a) If the <local or schema qualified name> is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

   b) If the <local or schema qualified name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

   c) Otherwise, the <schema name> that is specified or implicit for the SQL-client module is implicit.

5) Let *TN* be a <table name> with a <qualified identifier> *QI* and a <local or schema qualifier> *LSQ*.

Case:

a) If *LSQ* is "MODULE", then *TN* shall be contained in an <SQL-client module definition> *M* and the <module contents> of *M* shall contain a <temporary table declaration> *TT* whose <table name> has a <qualified identifier> equivalent to *QI*.

b) Otherwise, *LSQ* shall be a <schema name> that identifies a schema that contains a <table definition> or <view definition> whose <table name> has a <qualified identifier> equivalent to *QI*.

6) If a <cursor name> *CN* with a <qualified identifier> *QI* does not contain a <local qualifier>, then the <local qualifier> MODULE is implicit.

7) Let *CN* be a <cursor name> with a <qualified identifier> *QI* and a <local qualifier> *LQ*. *LQ* shall be "MODULE" and *CN* shall be contained in an <SQL-client module definition> whose <module contents> contain a <declare cursor> whose <cursor name> is *CN*.

8) If <user-defined type name> *UDTN* with a <qualified identifier> *QI* is specified, then

Case:

a) If *UDTN* is simply contained in <path-resolved user-defined type name>, then

Case:

i) If *UDTN* contains a <schema name> *SN*, then the schema identified by *SN* shall contain the descriptor of a user-defined type *UDT* such that the <qualified identifier> of *UDT* is equivalent to *QI*. *UDT* is the user-defined type identified by *UDTN*.

ii) Otherwise,

1) Case:

A) If *UDTN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then let *DP* be the SQL-path of the current SQL-session.

B) If *UDTN* is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

C) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains *UDTN*.

2) Let *N* be the number of <schema name>s in *DP*. Let S_i, 1 (one) $\leq i \leq N$, be the *i*-th <schema name> in *DP*.

3) Let the *set of subject types* be the set containing every user-defined type *T* in the schema identified by some S_i, 1 (one) $\leq i \leq N$, such that the <qualified identifier> of *T* is equivalent to *QI*. There shall be at least one type in the set of subject types.

4) Let *UDT* be the user-defined type contained in the set of subject types such that there is no other type *UDT2* for which the <schema name> of the schema that includes the user-defined type descriptor of *UDT2* precedes in *DP* the <schema name> identifying the schema that includes the user-defined type descriptor of *UDT*. *UDTN* identifies *UDT*.

5) The implicit <schema name> of *UDTN* is the <schema name> of the schema that includes the user-defined type descriptor of *UDT*.

b) If *UDTN* is simply contained in <schema-resolved user-defined type name> and *UDTN* does not contain a <schema name>, then

Case:

i) If *UDTN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the implicit <schema name> of *UDTN* is the default <unqualified schema name> for the SQL-session.

ii) If *UDTN* is contained in a <schema definition>, then the implicit <schema name> of *UDTN* is the <schema name> that is specified or implicit in <schema definition>.

iii) Otherwise, the implicit <schema name> of *UDTN* is the <schema name> that is specified or implicit in <SQL-client module definition>.

9) Two <user-defined type name>s are equivalent if any only if they have equivalent <qualified identifier>s and equivalent <schema name>s, regardless of whether the <schema name>s are implicit or explicit.

10) No <unqualified schema name> shall specify DEFINITION_SCHEMA.

11) If a <transcoding name> does not specify a <schema name>, then INFORMATION_SCHEMA is implicit; otherwise, INFORMATION_SCHEMA shall be specified.

12) If a <character set name> does not specify a <schema name>, then

Case:

a) If <character set name> is not immediately contained in:

i) A <character set definition>.

ii) A <drop character set statement>.

then <schema name> INFORMATION_SCHEMA is implicit.

b) Otherwise,

Case:

i) If the <character set name> is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

ii) If the <character set name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

iii) Otherwise, the <character set name> that is specified or implicit for the <SQL-client module definition> is implicit.

13) If a <schema qualified name> *SQN* other than a <transcoding name> does not contain a <schema name>, then

Case:

a) If any of the following is true:

i) *SQN* is is immediately contained in a <collation name> that is not immediately contained in a <collation definition> or in a <drop collation statement>.

ii) *SQN* is immediately contained in a <transliteration name> that is not immediately contained in a <transliteration definition> or in a <drop transliteration statement>.

then<schema name> INFORMATION_SCHEMA is implicit.

b) Otherwise,

Case:

i) If the <schema qualified name> is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

ii) If the <schema qualified name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

iii) Otherwise, the <schema name> that is specified or implicit for the <SQL-client module definition> is implicit.

14) If a <schema name> does not contain a <catalog name>, then

Case:

a) If the <unqualified schema name> is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default catalog name for the SQL-session is implicit.

b) If the <unqualified schema name> is contained in a <module authorization clause>, then an implementation-defined <catalog name> is implicit.

c) If the <unqualified schema name> is contained in a <schema definition> other than in a <schema name clause>, then the <catalog name> that is specified or implicit in the <schema name clause> is implicit.

d) If the <unqualified schema name> is contained in a <schema name clause>, then

Case:

i) If the <schema name clause> is contained in an <SQL-client module definition>, then the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.

ii) Otherwise, an implementation-defined <catalog name> is implicit.

e) Otherwise, the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.

15) Two <schema qualified name>s are equivalent if and only if their <qualified identifier>s are equivalent and their <schema name>s are equivalent, regardless of whether the <schema name>s are implicit or explicit.

16) Two <local or schema qualified name>s are equivalent if and only if their <qualified identifier>s are equivalent and either they both specify MODULE or they both specify or imply <schema name>s that are equivalent.

17) Two <character set name>s are equivalent if and only if their <SQL language identifier>s are equivalent and their <schema name>s are equivalent, regardless of whether the <schema name>s are implicit or explicit.

18) Two <schema name>s are equivalent if and only if their <unqualified schema name>s are equivalent and their <catalog name>s are equivalent, regardless of whether the <catalog name>s are implicit or explicit.

19) An <identifier> that is a <correlation name> is associated with a table within a particular scope. The scope of a <correlation name> is either a <select statement: single row>, <subquery>, or <query specification> (see Subclause 7.6, "<table reference>"), or is a <trigger definition> (see Subclause 11.39, "<trigger definition>"). Scopes may be nested. In different scopes, the same <correlation name> may be associated with different tables or with the same table.

20) No <authorization identifier> shall specify "PUBLIC".

21) Those <identifier>s that are valid <authorization identifier>s are implementation-defined.

22) Those <identifier>s that are valid <catalog name>s are implementation-defined.

23) The <data type> of <SQL-server name>, <connection name>, and <connection user name> shall be character string with an implementation-defined character set and shall have an octet length of 128 characters or less.

24) The <simple value specification> of <extended statement name> or <extended cursor name> shall not be a <literal>.

25) The declared type of the <simple value specification> of <extended statement name> shall be character string with an implementation-defined character set and shall have an octet length of 128 octets or less.

26) The declared type of the <simple value specification> of <extended cursor name> shall be character string with an implementation-defined character set and shall have an octet length of 128 octets or less.

27) The declared type of the <simple value specification> of <descriptor name> shall be character string with an implementation-defined character set and shall have an octet length of 128 octets or less.

28) In a <descriptor name>, <extended statement name>, or <extended cursor name>, if a <scope option> is not specified, then a <scope option> of LOCAL is implicit.

Access Rules

None.

General Rules

1) A <table name> identifies a table.

2) Within its scope, a <correlation name> identifies a table.

3) Within its scope, a <query name> identifies the table defined or returned by some associated <query expression body>.

4) A <column name> identifies a column.

5) A <domain name> identifies a domain.

6) An <authorization identifier> identifies a set of privileges.

7) An <SQL-client module name> identifies an SQL-client module.

8) A <schema qualified routine name> identifies an SQL-invoked routine.

9) A <method name> identifies an SQL-invoked method *M* whose descriptor is included in the schema that includes the descriptor of the user-defined type that is the type of *M*.

10) A <specific name> identifies an SQL-invoked routine.

11) A <cursor name> identifies a cursor.

12) A <host parameter name> identifies a host parameter.

13) An <SQL parameter name> identifies an SQL parameter.

14) An <external routine name> identifies an external routine.

15) A <trigger name> identifies a trigger.

16) A <constraint name> identifies a table constraint, a domain constraint, or an assertion.

17) A <catalog name> identifies a catalog.

18) A <schema name> identifies a schema.

19) A <collation name> identifies a collation.

20) A <character set name> identifies a character set.

21) A <transliteration name> identifies a character transliteration.

22) A <transcoding name> identifies a transcoding. All <transcoding name>s are implementation-defined.

23) A <connection name> identifies an SQL-connection.

24) A <user-defined type name> identifies a user-defined type.

25) An <attribute name> identifies an attribute of a structured type.

26) A <savepoint name> identifies a savepoint. The scope of a <savepoint name> is the SQL-transaction in which it was defined.

27) A <sequence generator name> identifies a sequence generator.

28) A <field name> identifies a field.

29) A <role name> identifies a role.

30) A <user identifier> identifies a user.

31) The value *ESN* of an <extended statement name> identifies a statement prepared by the execution of a <prepare statement>. If a <scope option> of GLOBAL is specified, then *ESN* is a global extended name; otherwise, it is a local extended name.

NOTE 78 — The scope of an extended name is defined in Subclause 4.24.2, "Dynamic SQL statements and descriptor areas".

32) A <dynamic cursor name> is a non-extended name that identifies a cursor in an <SQL dynamic statement>.

NOTE 79 — The scope of a non-extended name is defined in Subclause 4.24.2, "Dynamic SQL statements and descriptor areas".

33) A <statement name> is a non-extended name that identifies a prepared statement created by the execution of a <prepare statement>.

34) The value *ECN* of an <extended cursor name> identifies a cursor created by the execution of an <allocate cursor statement>. If a <scope option> of GLOBAL is specified, then *ECN* is a global extended name; otherwise, it is a local extended name.

35) The value *DN* of a <descriptor name> identifies an SQL descriptor area created by the execution of an <allocate descriptor statement>. If a <scope option> of GLOBAL is specified, then *DN* is a global extended name; otherwise, it is a local extended name.

36) A <window name> identifies a window.

Conformance Rules

1) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <savepoint name>.

2) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <role name>.

3) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", conforming SQL language shall not contain a <query name>.

4) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <attribute name>.

5) Without Feature T051, "Row types", conforming SQL language shall not contain a <field name>.

6) Without Feature F651, "Catalog name qualifiers", conforming SQL language shall not contain a <catalog name>.

7) Without Feature F771, "Connection management", conforming SQL language shall not contain an explicit <connection name>.

8) Without Feature F690, "Collation support", conforming SQL language shall not contain a <collation name>.

9) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transliteration name>.

10) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transcoding name>.

11) Without Feature F821, "Local table references", conforming SQL language shall not contain a <local or schema qualifier> that contains a <local qualifier>.

12) Without Feature F251, "Domain support", conforming SQL language shall not contain a <domain name>.

13) Without Feature F491, "Constraint management", conforming SQL language shall not contain a <constraint name>.

14) Without Feature F461, "Named character sets", conforming SQL language shall not contain a <character set name>.

15) Without Feature T601, "Local cursor references", a <cursor name> shall not contain a <local qualifier>.

16) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <extended statement name> or <extended cursor name>.

17) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <SQL statement name>.

18) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain <dynamic cursor name>.

19) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <descriptor name>.

20) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window name>.

21) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <sequence generator name>.

22) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <descriptor name> that is not a <literal>.

6 Scalar expressions

6.1 <data type>

Function

Specify a data type.

Format

```
<data type> ::=
    <predefined type>
  | <row type>
  | <path-resolved user-defined type name>
  | <reference type>
  | <collection type>

<predefined type> ::=
    <character string type> [ CHARACTER SET <character set specification> ]
    [ <collate clause> ]
  | <national character string type> [ <collate clause> ]
  | <binary large object string type>
  | <numeric type>
  | <boolean type>
  | <datetime type>
  | <interval type>

<character string type> ::=
    CHARACTER [ <left paren> <length> <right paren> ]
  | CHAR [ <left paren> <length> <right paren> ]
  | CHARACTER VARYING <left paren> <length> <right paren>
  | CHAR VARYING <left paren> <length> <right paren>
  | VARCHAR <left paren> <length> <right paren>
  | <character large object type>

<character large object type> ::=
    CHARACTER LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | CHAR LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | CLOB [ <left paren> <large object length> <right paren> ]

<national character string type> ::=
    NATIONAL CHARACTER [ <left paren> <length> <right paren> ]
  | NATIONAL CHAR [ <left paren> <length> <right paren> ]
  | NCHAR [ <left paren> <length> <right paren> ]
  | NATIONAL CHARACTER VARYING <left paren> <length> <right paren>
  | NATIONAL CHAR VARYING <left paren> <length> <right paren>
  | NCHAR VARYING <left paren> <length> <right paren>
  | <national character large object type>
```

```
<national character large object type> ::=
    NATIONAL CHARACTER LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | NCHAR LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | NCLOB [ <left paren> <large object length> <right paren> ]

<binary large object string type> ::=
    BINARY LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | BLOB [ <left paren> <large object length> <right paren> ]

<numeric type> ::=
    <exact numeric type>
  | <approximate numeric type>

<exact numeric type> ::=
    NUMERIC [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | DECIMAL [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | DEC [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | SMALLINT
  | INTEGER
  | INT
  | BIGINT

<approximate numeric type> ::=
    FLOAT [ <left paren> <precision> <right paren> ]
  | REAL
  | DOUBLE PRECISION

<length> ::= <unsigned integer> [ <char length units> ]

<large object length> ::=
    <unsigned integer> [ <multiplier> ] [ <char length units> ]
  | <large object length token> [ <char length units> ]

<char length units> ::=
    CHARACTERS
  | OCTETS

<precision> ::= <unsigned integer>

<scale> ::= <unsigned integer>

<boolean type> ::= BOOLEAN

<datetime type> ::=
    DATE
  | TIME [ <left paren> <time precision> <right paren> ] [ <with or without time zone> ]
  | TIMESTAMP [ <left paren> <timestamp precision> <right paren> ]
    [ <with or without time zone> ]

<with or without time zone> ::=
    WITH TIME ZONE
  | WITHOUT TIME ZONE

<time precision> ::= <time fractional seconds precision>

<timestamp precision> ::= <time fractional seconds precision>

<time fractional seconds precision> ::= <unsigned integer>
```

```
<interval type> ::= INTERVAL <interval qualifier>

<row type> ::= ROW <row type body>

<row type body> ::=
    <left paren> <field definition> [ { <comma> <field definition> }... ] <right paren>

<reference type> ::= REF <left paren> <referenced type> <right paren> [ <scope clause> ]

<scope clause> ::= SCOPE <table name>

<referenced type> ::= <path-resolved user-defined type name>

<path-resolved user-defined type name> ::= <user-defined type name>

<collection type> ::=
    <array type>
  | <multiset type>

<array type> ::=
    <data type> ARRAY
    [ <left bracket or trigraph> <maximum cardinality> <right bracket or trigraph> ]

<maximum cardinality> ::= <unsigned integer>

<multiset type> ::= <data type> MULTISET
```

Syntax Rules

1) CHAR is equivalent to CHARACTER. DEC is equivalent to DECIMAL. INT is equivalent to INTEGER. VARCHAR is equivalent to CHARACTER VARYING. NCHAR is equivalent to NATIONAL CHARACTER. CLOB is equivalent to CHARACTER LARGE OBJECT. NCLOB is equivalent to NATIONAL CHARACTER LARGE OBJECT. BLOB is equivalent to BINARY LARGE OBJECT.

2) "NATIONAL CHARACTER" is equivalent to the corresponding <character string type> with a specification of "CHARACTER SET *CSN*", where "*CSN*" is an implementation-defined <character set name>.

3) If <character string type> is specified, then the collation derivation of the resulting character string type is *implicit*.

   Case:

   a) If <collate clause> is specified, then the collation specified by it shall be applicable to the explicit or implicit character set *CS* of the character string type. That collation is the declared type collation of the character string type.

   b) Otherwise, the character set collation of *CS* is the declared type collation of the character string type.

4) The value of a <length> or a <precision> shall be greater than 0 (zero).

5) If <length> is omitted, then a <length> of 1 (one) is implicit.

6) If <char length units> is specified, then the character repertoire of the explicit or implicit character set of the character type shall be UCS.

7) If <char length units> is not specified, CHARACTERS is implicit.

8) If <large object length> is omitted, then an implementation-defined <large object length> is implicit.

9) The numeric value of a <large object length> is determined as follows.

Case:

a) If <large object length> immediately contains <unsigned integer> and does not immediately contain <multiplier>, then the numeric value of <large object length> is the numeric value of the specified <unsigned integer>.

b) If <large object length> immediately contains <large object length token> or immediately contains <unsigned integer> and <multiplier>, then let *D* be the value of the specified <unsigned integer> or the numeric value of the sequence of <digit>s of <large object length token> interpreted as an <unsigned integer>. The numeric value of <large object length> is the numeric value resulting from the multiplication of *D* and *MS*, then *MS* is:

i) If <multiplier> is K, then 1,024.

ii) If <multiplier> is M, then 1,048,576.

iii) If <multiplier> is G, then 1,073,741,824.

10) If a <scale> is omitted, then a <scale> of 0 (zero) is implicit.

11) If a <precision> is omitted, then an implementation-defined <precision> is implicit.

12) CHARACTER specifies the data type character string.

13) Characters in a character string are numbered beginning with 1 (one).

14) Case:

a) If neither VARYING nor LARGE OBJECT is specified in <character string type>, then the length in characters of the character string is fixed and is the value of <length>.

b) If VARYING is specified in <character string type>, then the length in characters of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <length>.

c) If LARGE OBJECT is specified in a <character string type>, then the length in characters of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.

15) The maximum values of <length> and <large object length> are implementation-defined. Neither <length> nor <large object length> shall be greater than the corresponding maximum value.

16) If <character string type> is not contained in a <domain definition> or a <column definition> and CHARACTER SET is not specified, then an implementation-defined <character set specification> that specifies an implementation-defined character set that contains at least every character that is in <SQL language character> is implicit.

NOTE 80 — Subclause 11.24, "<domain definition>", and Subclause 11.4, "<column definition>", specify the result when <character string type> is contained in a <domain definition> or <column definition>, respectively.

17) BINARY LARGE OBJECT specifies the data type binary string.

18) Octets in a binary large object string are numbered beginning with 1 (one). The length in octets of the string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.

19) The <scale> of an <exact numeric type> shall not be greater than the <precision> of the <exact numeric type>.

20) For the <exact numeric type>s DECIMAL and NUMERIC:

a) The maximum value of <precision> is implementation-defined. <precision> shall not be greater than this value.

b) The maximum value of <scale> is implementation-defined. <scale> shall not be greater than this maximum value.

21) NUMERIC specifies the data type exact numeric, with the decimal precision and scale specified by the <precision> and <scale>.

22) DECIMAL specifies the data type exact numeric, with the decimal scale specified by the <scale> and the implementation-defined decimal precision equal to or greater than the value of the specified <precision>.

23) SMALLINT, INTEGER, and BIGINT specify the data type exact numeric, with scale of 0 (zero) and binary or decimal precision. The choice of binary versus decimal precision is implementation-defined, but the same radix shall be chosen for all three data types. The precision of SMALLINT shall be less than or equal to the precision of INTEGER, and the precision of BIGINT shall be greater than or equal to the precision of INTEGER.

24) FLOAT specifies the data type approximate numeric, with binary precision equal to or greater than the value of the specified <precision>. The maximum value of <precision> is implementation-defined. <precision> shall not be greater than this value.

25) REAL specifies the data type approximate numeric, with implementation-defined precision.

26) DOUBLE PRECISION specifies the data type approximate numeric, with implementation-defined precision that is greater than the implementation-defined precision of REAL.

27) For the <approximate numeric type>s FLOAT, REAL, and DOUBLE PRECISION, the maximum and minimum values of the exponent are implementation-defined.

28) If <time precision> is not specified, then 0 (zero) is implicit. If <timestamp precision> is not specified, then 6 is implicit.

29) If <with or without time zone> is not specified, then WITHOUT TIME ZONE is implicit.

30) The maximum value of <time precision> and the maximum value of <timestamp precision> shall be the same implementation-defined value that is not less than 6. The values of <time precision> and <timestamp precision> shall not be greater than that maximum value.

31) The length of a DATE is 10 positions. The length of a TIME WITHOUT TIME ZONE is 8 positions plus the <time fractional seconds precision>, plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIME WITH TIME ZONE is 14 positions plus the <time fractional seconds precision> plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIMESTAMP WITHOUT TIME ZONE is 19 positions plus the <time fractional seconds precision>, plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIMESTAMP WITH TIME ZONE is 25 positions plus the <time fractional seconds precision> plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero).

32) An <interval type> specifying an <interval qualifier> whose <start field> and <end field> are both either YEAR or MONTH or whose <single datetime field> is YEAR or MONTH is a *year-month interval* type. An <interval type> that is not a year-month interval type is a *day-time interval* type.

NOTE 81 — The length of interval data types is specified in the General Rules of Subclause 10.1, "<interval qualifier>".

33) The *i*-th value of an interval data type corresponds to the *i*-th <primary datetime field>.

34) If <data type> is a <reference type>, then at least one of the following conditions shall be true:

a) There exists a user-defined type descriptor whose user-defined type name is <user-defined type name> *UDTN* simply contained in <referenced type>. *UDTN* shall identify a structured type.

b) <reference type> is contained in the <member list> of <user-defined type definition> *UDTD* and the <path-resolved user-defined type name> simply contained in <referenced type> is equivalent to the <schema-resolved user-defined type name> contained in *UDTD*.

35) The <table name> contained in a <scope clause> shall identify a referenceable table whose structured type is *UDTN*.

36) The <table name> *STN* specified in <scope clause> identifies the scope of the reference type. This scope consists of every row in the table identified by *STN*.

37) An <array type> *AT* specifies an *array type*. The <data type> immediately contained in *AT* is the *element type* of the array type. The <maximum cardinality> immediately contained in *AT* is the *maximum cardinality* of a site of data type *AT*. If the maximum cardinality is not specified, then an implementation-defined maximum cardinality is implicit.

38) A <multiset type> *MT* specifies a *multiset type*. The <data type> immediately contained in *MT* is the *element type* of the multiset type.

39) <row type> specifies the row data type.

40) BOOLEAN specifies the boolean data type.

41) If <data type> *DT1* is contained in a <data type> *DT2*, then the *root data type* of *DT1* is the outermost <data type> that contains *DT1*.

Access Rules

1) If <user-defined type name>, <reference type>, <row type>, or <collection type> *TY* is specified, and *TY* is usage-dependent on a user-defined type *UDT*, then

Case:

a) If *TY* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges shall include the USAGE privilege on *UDT*.

b) Otherwise, the current privileges shall include the USAGE privilege on *UDT*.

NOTE 82 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

General Rules

1) If the result of any specification or operation would be a character value one of whose characters is not in the character set of its declared type, then an exception condition is raised: *data exception — character not in repertoire.*

2) If any specification or operation attempts to cause an item of a character string type whose character set has a character repertoire of UCS to contain a code point that is a noncharacter, then an exception condition is raised: *data exception — noncharacter in UCS string.*

3) If <char length units> other than CHARACTERS is specified, then the conversion of the value of <length> to characters is implementation-defined.

4) For a <datetime type>,

   Case:

   a) If DATE is specified, then the data type contains the <primary datetime field>s years, months, and days.

   b) If TIME is specified, then the data type contains the <primary datetime field>s hours, minutes, and seconds.

   c) If TIMESTAMP is specified, then the data type contains the <primary datetime field>s years, months, days, hours, minutes, and seconds.

   d) If WITH TIME ZONE is specified, then the data type contains the time zone datetime fields.

      NOTE 83 — Within the non-null values of a <datetime type>, the value of the time zone interval is in the range –13:59 to +14:00. The range for time zone intervals is larger than many readers might expect because it is governed by political decisions in governmental bodies rather than by any natural law.

   NOTE 84 — A <datetime type> contains no other fields than those specified by the preceding Rule.

5) For a <datetime type>, a <time fractional seconds precision> that is an explicit or implicit <time precision> or <timestamp precision> defines the number of decimal digits following the decimal point in the SECOND <primary datetime field>.

6) Table 9, "Valid values for datetime fields", specifies the constraints on the values of the <primary datetime field>s in datetime values. The values of TIMEZONE_HOUR and TIMEZONE_MINUTE shall either both be non-negative or both be non-positive.

Table 9 — Valid values for datetime fields

Keyword	Valid values of datetime fields
YEAR	0001 to 9999
MONTH	01 to 12
DAY	Within the range 1 (one) to 31, but further constrained by the value of MONTH and YEAR fields, according to the rules for well-formed dates in the Gregorian calendar.

Keyword	Valid values of datetime fields
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 61.9(*N*) where "9(*N*)" indicates the number of digits specified by <time fractional seconds precision>.
TIMEZONE_HOUR	-12 to 14
TIMEZONE_MINUTE	-59 to 59

NOTE 85 — Datetime data types will allow dates in the Gregorian format to be stored in the date range 0001–01–01 CE through 9999–12–31 CE. The range for SECOND allows for as many as two "leap seconds". Interval arithmetic that involves leap seconds or discontinuities in calendars will produce implementation-defined results.

7) An interval value can be zero, positive, or negative.

8) The values of the <primary datetime field>s within an interval data type are constrained as follows:

   a) The value corresponding to the first <primary datetime field> is an integer with at most *N* digits, where *N* is the <interval leading field precision>.

   b) Table 10, "Valid absolute values for interval fields", specifies the constraints for the absolute values of other <primary datetime field>s in interval values.

   c) If an interval value is zero, then all fields of the interval are zero.

   d) If an interval value is positive, then all fields of the interval are non-negative and at least one field is positive.

   e) If an interval value is negative, then all fields of the interval are non-positive, and at least one field is negative.

Table 10 — Valid absolute values for interval fields

Keyword	Valid values of INTERVAL fields
MONTH	0 to 11
HOUR	0 to 23
MINUTE	0 to 59
SECOND	0 to 59.9(*N*) where "9(*N*)" indicates the number of digits specified by <interval fractional seconds precision> in the <interval qualifier>.

9) If <data type> specifies a character string type, then a character string type descriptor is created, including the following:

a) The name of the data type (either CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT).

b) The length or maximum length in characters of the character string type.

c) The catalog name, schema name, and character set name of the character set of the character string type.

d) The catalog name, schema name, and collation name of the collation of the character string type.

10) If <data type> is a <binary large object string type>, then a binary string data type descriptor is created, including the following:

a) The name of the data type (BINARY LARGE OBJECT).

b) The maximum length in octets of the binary string data type.

11) If <data type> *DT* specifies an exact numeric type, then:

a) There shall be an implementation-defined function *ENNF*() that converts any <exact numeric type> *ENT1* into some possibly different <exact numeric type> *ENT2* (the normal form of *ENT1*), subject to the following constraints on *ENNF*():

i) For every <exact numeric type> *ENT*, *ENNF*(*ENT*) shall not specify DEC or INT.

NOTE 86 — The preceding requirement prohibits the function *ENNF* from returning a value that uses the abbreviated spelling of the two data types; the function shall instead return the long versions of DECIMAL or INTEGER.

ii) For every <exact numeric type> *ENT*, the precision, scale, and radix of *ENNF*(*ENT*) shall be the precision, scale, and radix of *ENT*.

iii) For every <exact numeric type> *ENT*, *ENNF*(*ENT*) shall be the same as *ENNF*(*ENNF*(*ENT*)).

iv) For every <exact numeric type> *ENT*, if *ENNF*(*ENT*) specifies DECIMAL, then *ENNF*(*ENT*) shall specify <precision>, and the precision of *ENNF*(*ENT*) shall be the value of the <precision> specified in *ENNF*(*ENT*).

b) A numeric data type descriptor is created for *DT*, including the following:

i) The name of the type specified in *ENNF*(*DT*) (NUMERIC, DECIMAL, INTEGER, or SMALLINT).

ii) The precision of *DT*.

iii) The scale of *DT*.

iv) An indication of whether the precision and scale are expressed in decimal or binary terms.

12) If <data type> *DT* specifies an approximate numeric type, then:

a) There shall be an implementation-defined function *ANNF*() that converts any <approximate numeric type> *ANT* into some possibly different <approximate numeric type> *ANT2* (the normal form of *ANT1*), subject to the following constraints on *ANNF*():

i) For every <approximate numeric type> *ANT*, the precision of *ANNF*(*ANT*) shall be the precision of *ANT*.

ii) For every <approximate numeric type> *ANT*, *ANNF*(*ANT*) shall be the same as *ANNF*(*ANNF*(*ANT*)).

iii) For every <approximate numeric type> *ANT*, if *ANNF*(*ANT*) specifies FLOAT, then *ANNF*(*ANT*) shall specify <precision>, and the precision of *ANNF*(*ANT*) shall be the value of the <precision> specified in *ANNF*(*ANT*).

b) A numeric data type descriptor is created for *DT* including the following:

i) The name of the type specified in *ANNF*(*DT*) (FLOAT, REAL, or DOUBLE PRECISION).

ii) The precision of *DT*.

iii) An indication that the precision is expressed in binary terms.

13) If <data type> specifies <boolean type>, then a boolean data type descriptor is created, including the name of the boolean type (BOOLEAN).

14) If <data type> specifies a <datetime type>, then a datetime data type descriptor is created, including the following:

a) The name of the datetime type (DATE, TIME WITHOUT TIME ZONE, TIME WITH TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE).

b) The value of the <time fractional seconds precision>, if DATE is not specified.

15) If <data type> specifies an <interval type>, then an interval data type descriptor is created, including the following:

a) The name of the interval data type (INTERVAL).

b) An indication of whether the interval data type is a year-month interval or a day-time interval.

c) The <interval qualifier> simply contained in the <interval type>.

16) If <data type> is a <collection type>, then a collection type descriptor is created. Let *KC* be the kind of collection (either ARRAY or MULTISET) specified by <collection type>. Let *ET* be the element type of <collection type>. Let *ETD* be the type designator of *ET*. The collection type descriptor includes the type designator *EDT KC*, an indication of *KC*, the descriptor of *ET*, and (in the case of array types) the maximum cardinality.

17) For a <row type> *RT*, the degree of *RT* is initially set to 0 (zero). The General Rules of Subclause 6.2, "<field definition>", specify the degree of *RT* during the definition of the fields of *RT*.

18) If the <data type> is a <row type>, then a row type descriptor is created. The row type descriptor includes a field descriptor for every <field definition> of the <row type>.

19) A <reference type> identifies a reference type.

20) If <data type> is a <reference type>, then a reference type descriptor is created. Let *RDTN* be the name of the <referenced type>. The reference type descriptor includes the type designator REF(*RDTN*). If a <scope clause> is specified, then the reference type descriptor includes *STN*, identifying the scope of the reference type.

NOTE 87 — The user-defined type descriptor for a user-defined type is created in the General Rules of Subclause 11.41, "<user-defined type definition>".

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <path-resolved user-defined type name> that identifies a structured type.

2) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean type>.

3) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <time precision> that does not specify 0 (zero).

4) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <timestamp precision> that does not specify either 0 (zero) or 6.

5) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval type>.

6) Without Feature F421, "National character", conforming SQL language shall not contain a <national character string type>

7) Without Feature F411, "Time zone specification", conforming SQL language shall not contain <with or without time zone>.

8) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <reference type>.

9) Without Feature T051, "Row types", conforming SQL language shall not contain a <row type>.

10) Without Feature S091, "Basic array support", conforming SQL language shall not contain an <array type>.

11) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset type>.

12) Without Feature S281, "Nested collection types", conforming SQL language shall not contain a collection type that is based on a <data type> that contains a <collection type>.

13) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <scope clause> that is not simply contained in a <data type> that is simply contained in a <column definition>.

14) Without Feature S092, "Arrays of user-defined types", conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

15) Without Feature S272, "Multisets of user-defined types", conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

16) Without Feature S094, "Arrays of reference types", conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <reference type>.

17) Without Feature S274, "Multisets of reference types", conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <reference type>.

18) Without Feature S096, "Optional array bounds", conforming SQL language shall not contain an <array type> that does not immediately contain <maximum cardinality>.

19) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <binary large object string type>, a <character large object type>, or a <national character large object type>.

20) Without Feature T061, "UCS support", conforming SQL language shall not contain a <char length units>.

21) Without Feature T071, "BIGINT data type", conforming SQL language shall not contain BIGINT.

6.2 <field definition>

Function

Define a field of a row type.

Format

```
<field definition> ::= <field name> <data type>
```

Syntax Rules

1) Let *RT* be the <row type> that simply contains a <field definition>.

2) The <field name> shall not be equivalent to the <field name> of any other <field definition> simply contained in *RT*.

3) The declared type of the field is <data type>.

4) Let *DT* be the <data type>.

5) If *DT* is CHARACTER or CHARACTER VARYING and does not specify a <character set specification>, then the <character set specification> specified or implicit in the <schema character set specification> is implicit.

Access Rules

None.

General Rules

1) A data type descriptor is created that describes the declared type of the field being defined.

2) The degree of the row type *RT* being defined in the simply containing <row type> is increased by 1 (one).

3) A field descriptor is created that describes the field being defined. The field descriptor includes the following:

   a) The <field name>.

   b) The data type descriptor of the declared type of the field.

   c) The ordinal position of the field in *RT*.

4) The field descriptor is included in the row type descriptor for *RT*.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain a <field definition>.

6.3 <value expression primary>

Function

Specify a value that is syntactically self-delimited.

Format

```
<value expression primary> ::=
    <parenthesized value expression>
  | <nonparenthesized value expression primary>

<parenthesized value expression> ::= <left paren> <value expression> <right paren>

<nonparenthesized value expression primary> ::=
    <unsigned value specification>
  | <column reference>
  | <set function specification>
  | <window function>
  | <scalar subquery>
  | <case expression>
  | <cast specification>
  | <field reference>
  | <subtype treatment>
  | <method invocation>
  | <static method invocation>
  | <new specification>
  | <attribute or method reference>
  | <reference resolution>
  | <collection value constructor>
  | <array element reference>
  | <multiset element reference>
  | <routine invocation>
  | <next value expression>

<collection value constructor> ::=
    <array value constructor>
  | <multiset value constructor>
```

Syntax Rules

1) The declared type of a <value expression primary> is the declared type of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, or <next value expression>, or the effective returns type of the simply contained <routine invocation>, respectively.

2) Let *NVEP* be a <nonparenthesized value expression primary> of the form "*A.B C*", where *A* satisfies the Format of <schema name>, *B* satisfies the Format of <identifier>, and *C* satisfies the Format of <SQL

argument list>. If *NVEP* satisfies the Format, Syntax Rules, and Access Rules of Subclause 6.16, "<method invocation>", then *NVEP* is treated as a <method invocation>; otherwise, *NVEP* is treated as a <routine invocation>.

NOTE 88 — The formal grammar defined in the Format and Syntax Rules of Subclause 6.16, "<method invocation>", and of Subclause 10.4, "<routine invocation>", does not necessarily disambiguate between a <method invocation> and the invocation of a regular function. In such cases, the preceding Syntax Rule ensures that a <nonparenthesized value expression primary> that satisfies the Format, Syntax Rules, and Access Rules of Subclause 6.16, "<method invocation>", is treated as a <method invocation>.

3) The declared type of a <collection value constructor> is the declared type of the <array value constructor> or <multiset value constructor> that it immediately contains.

Access Rules

None.

General Rules

1) The value of a <value expression primary> is the value of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, <routine invocation>, or <next value expression>.

2) The value of a <collection value constructor> is the value of the <array value constructor> or <multiset value constructor> that it immediately contains.

Conformance Rules

None.

6.4 <value specification> and <target specification>

Function

Specify one or more values, host parameters, SQL parameters, dynamic parameters, or host variables.

Format

```
<value specification> ::=
    <literal>
  | <general value specification>

<unsigned value specification> ::=
    <unsigned literal>
  | <general value specification>

<general value specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <dynamic parameter specification>
  | <embedded variable specification>
  | <current collation specification>
  | CURRENT_DEFAULT_TRANSFORM_GROUP
  | CURRENT_PATH
  | CURRENT_ROLE
  | CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>
  | CURRENT_USER
  | SESSION_USER
  | SYSTEM_USER
  | USER
  | VALUE

<simple value specification> ::=
    <literal>
  | <host parameter name>
  | <SQL parameter reference>
  | <embedded variable name>

<target specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <column reference>
  | <target array element specification>
  | <dynamic parameter specification>
  | <embedded variable specification>

<simple target specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <column reference>
  | <embedded variable name>

<host parameter specification> ::= <host parameter name> [ <indicator parameter> ]
```

```
<dynamic parameter specification> ::= <question mark>

<embedded variable specification> ::= <embedded variable name> [ <indicator variable> ]

<indicator variable> ::= [ INDICATOR ] <embedded variable name>

<indicator parameter> ::= [ INDICATOR ] <host parameter name>

<target array element specification> ::=
    <target array reference>
    <left bracket or trigraph> <simple value specification> <right bracket or trigraph>

<target array reference> ::=
    <SQL parameter reference>
  | <column reference>

<current collation specification> ::=
    COLLATION FOR <left paren> <string value expression> <right paren>
```

Syntax Rules

1) The declared type of an <indicator parameter> shall be exact numeric with scale 0 (zero).

2) Each <host parameter name> shall be contained in an <SQL-client module definition>.

3) If USER is specified, then CURRENT_USER is implicit.

   NOTE 89 — In an environment where the SQL-implementation conforms to Core SQL, conforming SQL language that contains either:

   — A specified or implied <comparison predicate> that compares the <value specification> USER with a <value specification> other than USER, or

   — A specified or implied assignment in which the "value" (as defined in Subclause 9.2, "Store assignment") contains the <value specification> USER

   will become non-conforming in an environment where the SQL-implementation conforms to some SQL package that supports character internationalization, unless the character repertoire of the implementation-defined character set in that environment is identical to the character repertoire of SQL_IDENTIFIER.

4) The declared type of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, and CURRENT_PATH is character string. Whether the character string is fixed length or variable length, and its length if it is fixed length or maximum length if it is variable length, are implementation-defined. The character set of the character string is SQL_IDENTIFIER. The declared type collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

5) The declared type of <string value expression> simply contained in <current collation specification> shall be character string. The declared type of <current collation specification> is character string. Whether the character string is fixed length or variable length, and its length if fixed length or maximum length if variable length, are implementation-defined. The character set of the character string is SQL_IDENTIFIER. The collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

6) The <value specification> or <unsigned value specification> VALUE shall be contained in a <domain constraint>. The declared type of an instance of VALUE is the declared type of the domain to which that domain constraint belongs.

7) A <target specification> or <simple target specification> that is a <column reference> shall be a new transition variable column reference.

NOTE 90 — "new transition variable column reference" is defined in Subclause 6.6, "<identifier chain>".

8) If <target array element specification> is specified, then:

a) The declared type of the <target array reference> shall be an array type.

b) The declared type of a <target array element specification> is the element type of the specified <target array reference>.

c) The declared type of <simple value specification> shall be exact numeric with scale 0 (zero).

9) The declared type of an <indicator variable> shall be exact numeric with a scale of 0 (zero).

10) Each <embedded variable name> shall be contained in an <embedded SQL statement>.

11) Each <dynamic parameter specification> shall be contained in a <preparable statement> that is dynamically prepared in the current SQL-session through the execution of a <prepare statement>.

12) The declared type of CURRENT_DEFAULT_TRANSFORM_GROUP and of CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is a character string. Whether the character string is fixed length or variable length, and its length if fixed length or maximum length if variable length, are implementation-defined. The character set of the character string is SQL_IDENTIFIER. The declared type collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

Access Rules

None.

General Rules

1) A <value specification> or <unsigned value specification> specifies a value that is not selected from a table.

2) A <host parameter specification> identifies a host parameter or a host parameter and an indicator parameter in an <SQL-client module definition>.

3) A <target specification> specifies a host parameter, an output SQL parameter, the column of a new transition variable, a parameter used in a dynamically prepared statement, or a host variable that can be assigned a value.

4) If a <host parameter specification> contains an <indicator parameter> and the value of the indicator parameter is negative, then the value specified by the <host parameter specification> is null; otherwise, the value specified by a <host parameter specification> is the value of the host parameter identified by the <host parameter name>.

5) The value specified by a <literal> is the value represented by that <literal>.

6) The value specified by CURRENT_USER is

Case:

a) If there is a current user identifier, then the value of that current user identifier.

b) Otherwise, the null value.

7) The value specified by SESSION_USER is the value of the SQL-session user identifier.

8) The value specified by CURRENT_ROLE is

Case:

a) If there is a current role name, then the value of that current role name.

b) Otherwise, the null value.

9) The value specified by SYSTEM_USER is equal to an implementation-defined string that represents the operating system user who executed the SQL-client module that contains the externally-invoked procedure whose execution caused the SYSTEM_USER <general value specification> to be evaluated.

10) The value specified by CURRENT_PATH is a <schema name list> where <catalog name>s are <delimited identifier>s and the <unqualified schema name>s are <delimited identifier>s. Each <schema name> is separated from the preceding <schema name> by a <comma> with no intervening <space>s. The schemas referenced in this <schema name list> are those referenced in the SQL-path of the current SQL-session context, in the order in which they appear in that SQL-path.

11) The value specified by <current collation specification> is the name of the collation of the <string value expression>.

12) If a <simple value specification> evaluates to the null value, then an exception condition is raised: *data exception — null value not allowed.*

13) A <simple target specification> specifies a host parameter, an output SQL parameter, or a column of a new transition variable. A <simple target specification> can only be assigned a value that is not null.

14) If a <target specification> or <simple target specification> is assigned a value that is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string.*

15) A <dynamic parameter specification> identifies a parameter used by a dynamically prepared statement.

16) An <embedded variable specification> identifies a host variable or a host variable and an indicator variable.

17) If an <embedded variable specification> contains an <indicator variable> and the value of the indicator variable is negative, then the value specified by the <embedded variable specification> is null; otherwise, the value specified by a <embedded variable specification> is the value of the host variable identified by the <embedded variable name>.

18) The value specified by CURRENT_DEFAULT_TRANSFORM_GROUP is the character string that represents the default transform group name in the SQL-session context.

19) The value specified by CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is the character string that represents the transform group name associated with the data type specified by <path-resolved user-defined type name>.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <general value specification> that contains CURRENT_PATH.

2) Without Feature F251, "Domain support", conforming SQL language shall not contain a <general value specification> that contains VALUE.

3) Without Feature F321, "User authorization", conforming SQL language shall not contain a <general value specification> that contains CURRENT_USER, SYSTEM_USER, or SESSION_USER.

   NOTE 91 — Although CURRENT_USER and USER are semantically the same, without Feature F321, "User authorization", CURRENT_USER shall be specified as USER.

4) Without Feature T332, "Extended roles", conforming SQL language shall not contain CURRENT_ROLE.

5) Without Feature F611, "Indicator data types", in conforming SQL language, the specific declared types of <indicator parameter>s and <indicator variable>s shall be the same implementation-defined data type.

6) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <general value specification> that contains a <dynamic parameter specification>.

7) Without Feature S097, "Array element assignment", conforming SQL language shall not contain a <target array element specification>.

8) Without Feature S241, "Transform functions", conforming SQL language shall not contain CURRENT_DEFAULT_TRANSFORM_GROUP.

9) Without Feature S241, "Transform functions", conforming SQL language shall not contain CURRENT_TRANSFORM_GROUP_FOR_TYPE.

10) Without Feature F693, "SQL-session and client module collations", conforming SQL language shall not contain <current collation specification>.

6.5 <contextually typed value specification>

Function

Specify a value whose data type is to be inferred from its context.

Format

```
<contextually typed value specification> ::=
    <implicitly typed value specification>
  | <default specification>

<implicitly typed value specification> ::=
    <null specification>
  | <empty specification>

<null specification> ::= NULL

<empty specification> ::=
    ARRAY <left bracket or trigraph> <right bracket or trigraph>
  | MULTISET <left bracket or trigraph> <right bracket or trigraph>

<default specification> ::= DEFAULT
```

Syntax Rules

1) Where the element type *ET* is determined by the context in which *ES* appears, the declared type *DT* of an <empty specification> *ES* is

   Case:

   a) If *ES* simply contains ARRAY, then *ET* ARRAY[0].

   b) If *ES* simply contains MULTISET, then *ET* MULTISET.

   ES is effectively replaced by `CAST ( ES AS DT )`.

   NOTE 92 — In every such context, *ES* is uniquely associated with some expression or site of declared type *DT*, which thereby becomes the declared type of *ES*.

2) The declared type *DT* of a <null specification> *NS* is determined by the context in which *NS* appears. *NS* is effectively replaced by CAST (*NS* AS *DT*).

   NOTE 93 — In every such context, *NS* is uniquely associated with some expression or site of declared type *DT*, which thereby becomes the declared type of *NS*.

3) The declared type *DT* of a <default specification> *DS* is the declared type of a <default option> *DO* included in some site descriptor, determined by the context in which *DS* appears. *DS* is effectively replaced by CAST (*DO* AS *DT*).

   NOTE 94 — In every such context, *DS* is uniquely associated with some site of declared type *DT*, which thereby becomes the declared type of *DS*.

Access Rules

None.

General Rules

1) An <empty specification> specifies a collection whose cardinality is zero.

2) A <null specification> specifies the null value.

3) A <default specification> specifies the default value of some associated item.

Conformance Rules

1) Without Feature S091, "Basic array support", conforming SQL language shall not contain an <empty specification> that simply contains ARRAY.

2) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain an <empty specification> that simply contains MULTISET.

6.6 <identifier chain>

Function

Disambiguate a <period>-separated chain of identifiers.

Format

```
<identifier chain> ::= <identifier> [ { <period> <identifier> }... ]

<basic identifier chain> ::= <identifier chain>
```

Syntax Rules

1) Let *IC* be an <identifier chain>.

2) Let *N* be the number of <identifier>s immediately contained in *IC*.

3) Let I_i, 1 (one) ≤ i ≤ N, be the <identifier>s immediately contained in *IC*, in order from left to right.

4) Let $PIC_1 = I_1$. For each j between 2 and N, let $PIC_j = PIC_{j-1}$ <period> I_j. PIC_j is called the j-th *partial identifier chain* of *IC*.

5) Let *M* be the minimum of *N* and 4.

6) A column *C* is said to be *refinable* if the declared type of *C* is a row type or a structured type.

7) An SQL parameter *P* is said to be *refinable* if the declared type of *P* is a row type or a structured type.

8) For at most one j between 1 (one) and *M*, PIC_j is called the *basis* of *IC*, and j is called the *basis length* of *IC*. The *referent* of the basis is a column *C* of a table or an SQL parameter *SP*. The basis, basis length, basis scope, and basis referent of *IC* are determined as follows:

   a) If $N = 1$ (one), then

      Case:

      i) If *IC* is contained in an <order by clause> of a <cursor specification>, and the <select list> simply contained in the <cursor specification> directly contains a <derived column> *DC* whose explicit or implicit <column name> is equivalent to *IC*, then PIC_1 is a candidate basis, the scope of PIC_1 is the <cursor specification>, and the referent of PIC_1 is the column referenced by *DC*.

      ii) Otherwise, *IC* shall be contained in the scope of one or more range variables whose associated tables include a column whose <column name> is equivalent to I_1 or in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter whose <SQL parameter name> is equivalent to I_1. Let the phrase *possible scope tags* denote those range variables and <routine name>s.

      NOTE 95 — "range variable" is defined in Subclause 4.14.6, "Operations involving tables".

      Case:

1) If the number of possible scope tags in the innermost scope containing a possible scope tag is 1 (one), then let *IPST* be that possible scope tag.

Case:

A) If *IPST* is a range variable *RV*, then let *T* be the table associated with *RV*. For every column *C* of *T* whose <column name> is equivalent to I_1, PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *RV*, and the referent of PIC_1 is *C*.

NOTE 96 — Two or more columns with equivalent column names are distinguished by their ordinal positions within *T*.

B) If the innermost possible scope tag is a <routine name>, then let *SP* be the SQL parameter whose <SQL parameter name> is equivalent to I_1. PIC_1 is the basis of *IC*, the basis length is 1 (one), the basis scope is the scope of *SP*, and the basis referent is *SP*.

2) Otherwise, each possible scope tag shall be a range variable *RV* of a <table factor> that is directly contained in a <joined table> *JT*. I_1 shall be a common column name in *JT*. Let *C* be the column of *JT* that is identified by I_1. PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *RV*, and the referent of PIC_1 is *C*.

NOTE 97 — "Common column name" is defined in Subclause 7.7, "<joined table>".

b) If $N > 1$ (one), then the basis, basis length, basis scope, and basis referent are defined in terms of a candidate basis as follows:

i) If *IC* is contained in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_1, then PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *SP*, and the referent of PIC_1 is *SP*.

ii) If $N = 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iii) If $N > 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes a refinable SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iv) If $N = 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. For every column *C* in the table associated with *EN* whose <column name> is equivalent to I_2, PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *EN*, and the referent of PIC_2 is *C*.

v) If $N > 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. For every refinable column *C* in the table associated with *EN* whose <column name> is equivalent to I_2, PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *EN*, and the referent of PIC_2 is *C*.

vi) If $N = 2$, 3 or 4, and if PIC_{N-1} is equivalent to an exposed <table or query name> that is in scope, then let *EN* be the exposed <table or query name> that is equivalent to PIC_{N-1} and has the innermost scope. For every column *C* in the table associated with *EN* whose <column name> is equivalent to I_N, PIC_N is a candidate basis of *IC*, the scope of PIC_N is the scope of *EN*, and the referent of PIC_N is *C*.

c) There shall be exactly one candidate basis *CB* with innermost scope. The basis of *IC* is *CB*. The basis length is the length of *CB*. The basis scope is the scope of *CB*. The referent of *IC* is the referent of *CB*.

9) Let *BL* be the basis length of *IC*.

10) If $BL < N$, then let *TIC* be the <value expression primary>:

(PIC_{BL}) <period> I_{BL+1} <period> ... <period> I_N

The Syntax Rules of Subclause 6.25, "<value expression>", are applied to *TIC*, yielding a column reference or an SQL parameter reference, and $(N - BL)$ <field reference>s or <method invocation>s.

NOTE 98 — In this transformation, (PIC_{BL}) is interpreted as a <value expression primary> of the form <left paren> <value expression> <right paren>. PIC_{BL} is a <value expression> that is a <value expression primary> that is an <unsigned value specification> that is either a <column reference> or an <SQL parameter reference>. The identifiers I_{BL+1}, ..., I_N are parsed using the Syntax Rules of <field reference> and <method invocation>.

11) A <basic identifier chain> shall be an <identifier chain> whose basis is the entire identifier chain.

12) A <basic identifier chain> whose basis referent is a column is a *column reference*. If the basis length is 2, and the basis scope is a <trigger definition> whose <trigger action time> is BEFORE, and I_1 is equivalent to the <new transition variable name> of the <trigger definition>, then the column reference is a *new transition variable column reference.*

13) A <basic identifier chain> whose basis referent is an SQL parameter is an *SQL parameter reference.*

14) The data type of a <basic identifier chain> *BIC* is the data type of the basis referent of *BIC*.

15) If the declared type of a <basic identifier chain> *BIC* is character string, then the collation derivation of the declared type of *BIC* is

Case:

a) If the declared type has a declared type collation *DTC*, then *implicit*.

b) Otherwise, *none*.

Access Rules

None.

General Rules

1) Let *BIC* be a <basic identifier chain>.

2) If *BIC* is a column reference, then *BIC* references the column *C* that is the basis referent of *BIC*.

3) If *BIC* is an SQL parameter reference, then *BIC* references the SQL parameter *SP* of a given invocation of the SQL-invoked routine that contains *SP*.

Conformance Rules

1) Without Feature T325, "Qualified SQL parameter references", conforming SQL language shall not contain an SQL parameter reference whose first <identifier> is the <qualified identifier> of a <routine name>.

6.7 <column reference>

Function

Reference a column.

Format

```
<column reference> ::=
    <basic identifier chain>
  | MODULE <period> <qualified identifier> <period> <column name>
```

Syntax Rules

1) Every <column reference> has a qualifying table and a qualifying scope, as defined in succeeding Syntax Rules.

2) A <column reference> that is a <basic identifier chain> *BIC* shall be a column reference. The qualifying scope is the basis scope of *BIC* and the qualifying table is the table that contains the basis referent of *BIC*.

3) If MODULE is specified, then <qualified identifier> shall be contained in an <SQL-client module definition> *M*, and shall identify a declared local temporary table *DLTT* whose <temporary table declaration> is contained in *M*, and "MODULE <period> <qualified identifier>" shall be an exposed <table or query name> *MPQI*, and <column name> shall identify a column of *DLTT*. The qualifying table is the table identified by *MPQI*, and the qualifying scope is the scope of *MPQI*.

4) If a <column reference> *CR* is contained in a <table expression> *TE* and the qualifying scope of *CR* contains *TE*, then *CR* is an *outer reference* to the qualifying table of *CR*.

5) Let *C* be the column that is referenced by *CR*. The declared type of *CR* is

   Case:

   a) If the column descriptor of *C* includes a data type,then that data type.

   b) Otherwise, the data type identified in the domain descriptor that is included in the column descriptor of *C*.

6) A column reference contained in a <query specification> or a <joined table> is a *queried column reference*.

7) If *QCR* is a queried column reference, then:

   a) The *qualifying query* of *QCR* is defined as follows.

      Case:

      i) If *QCR* is contained without an intervening <query specification> in a <joined table> *JT* that is a <query primary>, then *JT* is the qualifying query of *QCR*.

      ii) Otherwise, the <query specification> that simply contains the <from clause> that simply contains the <table reference> that defines the qualifying table of *QCR* is the qualifying query of *QCR*.

b) Let *QQ* be the qualifying query of *QCR*.

Case:

i) If *QQ* is a <joined table>, or if *QQ* is not grouped, or if *QCR* is contained in the <where clause> simply contained in *QQ*, then *QCR* is an *ordinary column reference*.

ii) If *QCR* is contained in the <having clause>, <window clause>, or <select list> simply contained in *QQ*, and *QCR* is contained in an aggregated argument of a <set function specification> *SFS*, and *QQ* is the aggregation query of *SFS*, then *QCR* is a *within-group-varying column reference*.

iii) Otherwise, *QCR* is a *group-invariant column reference*.

8) If *QCR* is a group-invariant column reference, then *QCR* shall be functionally dependent on the grouping columns of the qualifying query of *QCR*.

Access Rules

1) Let *CR* be the <column reference>.

2) If the qualifying table of *CR* is a base table or a viewed table, then

Case:

a) If *CR* is contained in a <search condition> immediately contained in an <assertion definition> or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on the column referenced by *CR*.

b) Otherwise,

Case:

i) If *CR* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on the column referenced by *CR*.

ii) Otherwise, the current privileges shall include SELECT on the column referenced by *CR*.

NOTE 99 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

General Rules

1) Let *QCR* be a queried column reference. Let *QT* be the qualifying table of *QCR*, and let *C* be the column of *QT* that is referenced as the basis referent of *QCR*. The value of *QCR* is determined as follows:

a) If *QCR* is an ordinary column reference, then *QCR* denotes the value of *C* in a given row of *QT*.

b) If *QCR* is a within-group-varying column reference, then *QCR* denotes the values of *C* in the rows of a given group of the qualifying query of *QCR* used to construct the argument source of a <set function specification>.

c) If *QCR* is a group-invariant column reference, then *QCR* denotes a value that is not distinct from the value of *C* in every row of a given group of the qualifying query of *QCR*. If the most specific type of

QCR is character string, datetime with time zone, or user-defined type, then the precise value is chosen in an implementation-dependent fashion.

Conformance Rules

1) Without Feature F821, "Local table references", conforming SQL language shall not contain a <column reference> that simply contains MODULE.

6.8 <SQL parameter reference>

Function

Reference an SQL parameter.

Format

```
<SQL parameter reference> ::= <basic identifier chain>
```

Syntax Rules

1) An <SQL parameter reference> shall be a <basic identifier chain> that is an SQL parameter reference.

2) The declared type of an <SQL parameter reference> is the declared type of the SQL parameter that it references.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

6.9 <set function specification>

Function

Specify a value derived by the application of a function to an argument.

Format

```
<set function specification> ::=
    <aggregate function>
  | <grouping operation>

<grouping operation> ::=
    GROUPING <left paren> <column reference>
    [ { <comma> <column reference> }... ] <right paren>
```

Syntax Rules

1) If <aggregate function> specifies a <general set function>, then the <value expression> simply contained in the <general set function> shall not contain a <set function specification> or a <subquery>.

2) If <aggregate function> specifies <binary set function>, then neither the <dependent variable expression> nor the <independent variable expression> simply contained in the <binary set function> shall contain a <set function specification> or a <subquery>.

3) A <value expression> *VE* simply contained in a <set function specification> *SFE* is an *aggregated argument* of *SFE* if either *SFE* is not an <ordered set function> or *VE* is simply contained in a <within group specification>; otherwise, *VE* is a *non-aggregated argument* of *SFE*.

4) A column reference *CR* contained in an aggregated argument of a <set function specification> *SFS* is called an *aggregated column reference* of *SFS*.

5) If <aggregate function> specifies a <filter clause>, then the <search condition> immediately contained in <filter clause> shall not contain a <set function specification>.

6) The *aggregation query* of a <set function specification> *SFS* is determined as follows.

   Case:

   a) If *SFS* has no aggregated column reference, then the aggregation query of *SFS* is the innermost <query specification> that contains *SFS*.

   b) Otherwise, the innermost qualifying query of the aggregated column references of *SFS* is the aggregation query of *SFS*.

7) *SFS* shall be contained in the <having clause>, <window clause>, or <select list> of its aggregation query.

8) Let *CR* be an aggregated column reference of *SFS* such that the qualifying query *QQ* of *CR* is not the aggregation query of *SFS*. If *QQ* is grouped and *SFS* is contained in the <having clause>, <window clause>, or <select list> of *QQ*, then *CR* shall be functionally dependent on the grouping columns of *QQ*.

9) If <aggregate function> is specified, then the declared type of the result is the declared type of the <aggregate function>.

10) If a <grouping operation> is specified, then:

a) Each <column reference> shall reference a grouping column of *T*.

b) The declared type of the result is exact numeric with an implementation-defined precision and a scale of 0 (zero).

c) If more than one <column reference> is specified, then let *N* be the number of <column reference>s and let CR_i, 1 (one) ≤ *i* ≤ *N*, be the *i*-th <column reference>.

GROUPING (CR_1, ..., CR_{N-1}, CR_N)

is equivalent to:

(2 * GROUPING (CR_1, ..., CR_{N-1}) + GROUPING (CR_N))

Access Rules

None.

General Rules

1) If <aggregate function> is specified, then the result is the value of the <aggregate function>.

NOTE 100 — The value of <grouping operation> is computed by means of syntactic transformations defined in Subclause 7.9, "<group by clause>".

Conformance Rules

1) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <grouping operation>.

2) Without Feature T433, "Multiargument GROUPING function", conforming SQL language shall not contain a <grouping operation> that contains more than one <column reference>.

6.10 <window function>

Function

Specify a window function.

Format

```
<window function> ::= <window function type> OVER <window name or specification>

<window function type> ::=
    <rank function type> <left paren> <right paren>
  | ROW_NUMBER <left paren> <right paren>
  | <aggregate function>

<rank function type> ::=
    RANK
  | DENSE_RANK
  | PERCENT_RANK
  | CUME_DIST

<window name or specification> ::=
    <window name>
  | <in-line window specification>

<in-line window specification> ::= <window specification>
```

Syntax Rules

1) An <aggregate function> simply contained in a <window function> shall not simply contain a <hypothetical set function>.

2) Let *OF* be the <window function>.

3) Case:

   a) If *OF* is contained in an <order by clause>, then the <order by clause> shall be contained in a <cursor specification> that is a simple table query. Let *ST* be the sort table that is obtained by applying the syntactic transformation of a simple table query, as specified in Subclause 14.1, "<declare cursor>". Let *TE* be the <table expression> contained in the result of that syntactic transformation.

   b) Otherwise, *OF* shall be contained in a <select list> that is immediately contained in a <query specification> *QS* or a <select statement: single row> *SSSR*. Let *QSS* be the innermost <query specification> contained in *QS* that contains *OF*. Let *TE* be the <table expression> immediately contained in *QSS* or *SSSR*.

4) *OF* shall not contain an outer reference or a <subquery>.

5) Let *WNS* be the <window name or specification>. Let *WDX* be a window structure descriptor that describes the window defined by *WNS*.

6) If <rank function type> or ROW_NUMBER is specified, then:

a) If RANK or DENSE_RANK is specified, then the window ordering clause *WOC* of *WDX* shall be present.

b) The window framing of *WDX* shall not be present.

c) Case:

   i) If *WNS* is a <window name>, then let *WNS1* be *WNS*.

   ii) Otherwise, let *WNS1* be the <window specification details> contained in *WNS*.

d) RANK ( ) OVER *WNS* is equivalent to:

```
( COUNT (*) OVER (WNS1 RANGE UNBOUNDED PRECEDING)
- COUNT (*) OVER (WNS1 RANGE CURRENT ROW) + 1 )
```

e) If DENSE_RANK is specified, then:

   i) Let VE_1, ..., VE_N be an enumeration of the <value expression>s that are <sort key>s simply contained in *WOC*.

   ii) DENSE_RANK ( ) OVER *WNS* is equivalent to the <window function>:

```
COUNT (DISTINCT ROW ( VE1, ..., VEN ) )
OVER (WNS1 RANGE UNBOUNDED PRECEDING)
```

f) ROW_NUMBER ( ) OVER *WNS* is equivalent to the <window function>:

```
COUNT (*) OVER (WNS1 ROWS UNBOUNDED PRECEDING)
```

g) Let *ANT1* be an approximate numeric type with implementation-defined precision. PERCENT_RANK ( ) OVER *WNS* is equivalent to:

```
CASE
  WHEN COUNT(*) OVER (WNS1 RANGE BETWEEN UNBOUNDED PRECEDING
                                    AND UNBOUNDED FOLLOWING) = 1
  THEN CAST (0 AS ANT1)
  ELSE
       (CAST (RANK () OVER (WNS1) AS ANT1) - 1) /
       (COUNT (*) OVER (WNS1 RANGE BETWEEN UNBOUNDED PRECEDING
                                AND UNBOUNDED FOLLOWING) - 1)
END
```

h) Let *ANT2* be an approximate numeric type with implementation-defined precision. CUME_DIST ( ) OVER *WNS* is equivalent to:

```
( CAST ( COUNT (*) OVER
     ( WNS1 RANGE UNBOUNDED PRECEDING ) AS ANT2 ) /
   COUNT(*) OVER ( WNS1 RANGE BETWEEN UNBOUNDED PRECEDING
                                  AND UNBOUNDED FOLLOWING ) )
```

7) Let *SL* be the <select list> that simply contains *OF*.

NOTE 101 — If *OF* is originally contained in an <order by clause> of a cursor that is a simple table query, the syntactic transformation of Subclause 14.1, "<declare cursor>", shall be applied prior to this rule.

8) Let *SQ* be the <set quantifier> of the <query specification> or <select statement: single row> that simply contains *SL*. If there is no <set quantifier>, then let *SQ* be a zero-length string.

9) If <in-line window specification> is specified, then:

   a) Let *WS* be the <window specification>.

   b) Let *WSN* be an implementation-dependent <window name> that is not equivalent to any other <window name> in the <table expression> or <select statement: single row> that simply contains *WS*.

   c) Let *OFT* be the <window function type>.

   d) Let *SLNEW* be the <select list> that is obtained from *SL* by replacing *OF* by:

      ```
      OFT OVER WSN
      ```

   e) Let *FC*, *WC*, *GBC*, and *HC* be <from clause>, <where clause>, <group by clause>, and <having clause>, respectively, of *TE*. If any of <where clause>, <group by clause>, or <having clause> is missing, then let *WC*, *GBC*, or *HC*, respectively, be a zero-length string.

   f) Case:

      i) If there is no <window clause> simply contained in *TE*, then let *WICNEW* be:

         ```
         WINDOW WSN AS WS
         ```

      ii) Otherwise, let *WIC* be the <window clause> simply contained in *TE* and let *WICNEW* be:

         ```
         WIC, WSN AS WS
         ```

   g) Let *TENEW* be:

      ```
      FC WC GBC HC WICNEW
      ```

   h) Case:

      i) If *OF* is simply contained in a <query specification>, then that <query specification> is equivalent to:

         ```
         SELECT SQ SLNEW TENEW
         ```

      ii) Otherwise, *OF* is simply contained in a <select statement: single row>. Let *STL* be the <select target list> of that <select statement: single row>. The <select statement: single row> is equivalent to:

         ```
         SELECT SQ SLNEW INTO STL TENEW
         ```

10) If the window ordering clause or the window framing clause of the window structure descriptor that describes the <window name or specification> is present, then no <aggregate function> simply contained in <window function> shall specify DISTINCT or <ordered set function>.

Access Rules

None.

General Rules

1) The value of <window function> is the value of the <aggregate function>.

Conformance Rules

1) Without Feature T611, "Elementary OLAP operations", conforming SQL language shall not contain a <window function>.

2) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window name>.

3) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain PERCENT_RANK or CUME_DIST.

4) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window function> that simply contains ROW_NUMBER and immediately contains a <window name or specification> whose window structure descriptor does not contain a window ordering clause.

6.11 <case expression>

Function

Specify a conditional value.

Format

```
<case expression> ::=
    <case abbreviation>
  | <case specification>

<case abbreviation> ::=
    NULLIF <left paren> <value expression> <comma> <value expression> <right paren>
  | COALESCE <left paren> <value expression>
    { <comma> <value expression> }... <right paren>

<case specification> ::=
    <simple case>
  | <searched case>

<simple case> ::= CASE <case operand> <simple when clause>... [ <else clause> ] END

<searched case> ::= CASE <searched when clause>... [ <else clause> ] END

<simple when clause> ::= WHEN <when operand> THEN <result>

<searched when clause> ::= WHEN <search condition> THEN <result>

<else clause> ::= ELSE <result>

<case operand> ::=
    <row value predicand>
  | <overlaps predicate part 1>

<when operand> ::=
    <row value predicand>
  | <comparison predicate part 2>
  | <between predicate part 2>
  | <in predicate part 2>
  | <character like predicate part 2>
  | <octet like predicate part 2>
  | <similar predicate part 2>
  | <null predicate part 2>
  | <quantified comparison predicate part 2>
  | <normalized predicate part 2>
  | <match predicate part 2>
  | <overlaps predicate part 2>
  | <distinct predicate part 2>
  | <member predicate part 2>
  | <submultiset predicate part 2>
  | <set predicate part 2>
  | <type predicate part 2>
```

```
<result> ::=
    <result expression>
  | NULL

<result expression> ::= <value expression>
```

Syntax Rules

1) If a <case specification> specifies a <case abbreviation>, then:

   a) A <value expression> generally contained in the <case abbreviation> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

   b) NULLIF (V_1, V_2) is equivalent to the following <case specification>:

      CASE WHEN V_1=V_2 THEN NULL ELSE V_1 END

   c) COALESCE (V_1, V_2) is equivalent to the following <case specification>:

      CASE WHEN V_1 IS NOT NULL THEN V_1 ELSE V_2 END

   d) COALESCE (V_1, V_2, ..., V_n), for $n \geq 3$, is equivalent to the following <case specification>:

      CASE WHEN V_1 IS NOT NULL THEN V_1 ELSE COALESCE (V_2, ..., V_n) END

2) If a <case specification> specifies a <simple case>, then let *CO* be the <case operand>.

   a) *CO* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

   b) If *CO* is <overlaps predicate part 1>, then each <when operand> shall be <overlaps predicate part 2>. If *CO* is <row value predicand>, then no <when operand> shall be an <overlaps predicate part 2>.

   c) If the <when operand> of the *i*-th <simple when clause> is a <row value predicand>, then let WO_i be the <equals operator> concatenated with the <when operand> of the *i*-th <simple when clause>; otherwise, let WO_i be the <when operand> of the *i*-th <simple when clause>.

   d) Let R_i be the <result> of the *i*-th <simple when clause>.

   e) If <else clause> is specified, then let *CEEC* be that <else clause>; otherwise, let *CEEC* be a character string of length 0 (zero).

   f) The <simple case> is equivalent to a <searched case> in which the *i*-th <searched when clause> takes the form:

      WHEN *CO* WO_i THEN R_i

   g) The <else clause> of the equivalent <searched case> takes the form:

      CEEC

3) At least one <result> in a <case specification> shall specify a <result expression>.

4) If an <else clause> is not specified, then ELSE NULL is implicit.

5) The declared type of a <case specification> is determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of all <result expression>s in the <case specification>.

Access Rules

None.

General Rules

1) Case:

   a) If a <result> specifies NULL, then its value is the null value.

   b) If a <result> specifies a <value expression>, then its value is the value of that <value expression>.

2) Case:

   a) If the <search condition> of some <searched when clause> in a <case specification> is *True*, then the value of the <case specification> is the value of the <result> of the first (leftmost) <searched when clause> whose <search condition> is *True*, cast as the declared type of the <case specification>.

   b) If no <search condition> in a <case specification> is *True*, then the value of the <case expression> is the value of the <result> of the explicit or implicit <else clause>, cast as the declared type of the <case specification>.

Conformance Rules

1) Without Feature F262, "Extended CASE expression", in conforming SQL language, a <case operand> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

2) Without Feature F262, "Extended CASE expression", in conforming SQL language, a <when operand> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

6.12 <cast specification>

Function

Specify a data conversion.

Format

```
<cast specification> ::= CAST <left paren> <cast operand> AS <cast target> <right paren>

<cast operand> ::=
    <value expression>
  | <implicitly typed value specification>

<cast target> ::=
    <domain name>
  | <data type>
```

Syntax Rules

1) Case:

   a) If a <domain name> is specified, then let *TD* be the <data type> of the specified domain.

   b) If a <data type> is specified, then let *TD* be the specified <data type>.

2) The declared type of the result of the <cast specification> is *TD*.

3) If the <cast operand> is a <value expression>, then let *SD* be the declared type of the <value expression>.

4) Let *C* be some column and let *CO* be the <cast operand> of a <cast specification> *CS*. *C* is a *leaf column* of *CS* if *CO* consists of a single column reference that identifies *C* or of a single <cast specification> *CS1* of which *C* is a leaf column.

5) If the <cast operand> specifies an <empty specification>, then *TD* shall be a collection type.

6) If the <cast operand> is a <value expression>, then the valid combinations of *TD* and *SD* in a <cast specification> are given by the following table. "Y" indicates that the combination is syntactically valid without restriction; "M" indicates that the combination is valid subject to other Syntax Rules in this Subclause being satisfied; and "N" indicates that the combination is not valid:

<data type> *SD* of <value expression>	<data type> of *TD*															
	EN	AN	VC	FC	D	T	TS	YM	DT	BO	UDT	CL	BL	RT	CT	RW
EN	Y	Y	Y	Y	N	N	N	M	M	N	M	Y	N	M	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N	M	Y	N	M	N	N
C	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	M	Y	N	M	N	N
D	N	N	Y	Y	Y	N	Y	N	N	N	M	Y	N	M	N	N
T	N	N	Y	Y	N	Y	Y	N	N	N	M	Y	N	M	N	N
TS	N	N	Y	Y	Y	Y	Y	N	N	N	M	Y	N	M	N	N

YM	M	N	Y	Y	N	N	N	Y	N	N	M	Y	N	M	N	N
DT	M	N	Y	Y	N	N	N	N	Y	N	M	Y	N	M	N	N
BO	N	N	Y	Y	N	N	N	N	N	Y	M	Y	N	M	N	N
UDT	M	M	M	M	M	M	M	M	M	M	M	M	M	M	N	N
BL	N	N	N	N	N	N	N	N	N	N	M	N	Y	M	N	N
RT	M	M	M	M	M	M	M	M	M	M	M	M	M	M	N	N
CT	N	N	N	N	N	N	N	N	N	N	N	N	N	N	M	N
RW	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	M

Where:

EN = Exact Numeric
AN = Approximate Numeric
C = Character (Fixed- or Variable-length, or character large object)
FC = Fixed-length Character
VC = Variable-length Character
CL = Character Large Object
D = Date
T = Time
TS = Timestamp
YM = Year-Month Interval
DT = Day-Time Interval
BO = Boolean
UDT = User-Defined Type
BL = Binary Large Object
RT = Reference type
CT = Collection type
RW = Row type

7) If *TD* is an interval and *SD* is exact numeric, then *TD* shall contain only a single <primary datetime field>.

8) If *TD* is exact numeric and *SD* is an interval, then *SD* shall contain only a single <primary datetime field>.

9) If *SD* is character string and *TD* is fixed-length, variable-length, or large object character string, then the character repertoires of *SD* and *TD* shall be the same.

10) If *TD* is a fixed-length, variable-length, or large object character string, then *TD* shall not specify <collate clause>. The declared type collation of the <cast specification> is the character set collation of the character set of *TD* and its collation derivation is *implicit*.

11) If the <cast operand> is a <value expression> and either *SD* or *TD* is a user-defined type, then either *TD* shall be a supertype of *SD* or there shall be a data type *P* such that:

    a) The type designator of *P* is in the type precedence list of *SD*.

    b) There is a user-defined cast CF_P whose user-defined cast descriptor includes *P* as the source data type and *TD* as the target data type.

    c) The type designator of no other data type *Q* that is included as the source data type in the user-defined cast descriptor of some user-defined cast CF_Q that has *TD* as the target data type precedes the type designator of *P* in the type precedence list of *SD*.

12) If the <cast operand> is a <value expression> and either *SD* or *TD* is a reference type, then:

    a) Let *RTSD* and *RTTD* be the referenced types of *SD* and *TD*, respectively.

b) If <data type> is specified and contains a <scope clause>, then let *STD* be that scope. Otherwise, let *STD*, possibly empty, be the scope included in the reference type descriptor of *SD*.

c) Either *RSTD* and *RTTD* shall be compatible, or there shall be a data type *P* in the type precedence list of *SD* such that all of the following are satisfied:

i) There is a user-defined cast CF_P whose user-defined cast descriptor includes *P* as the source data type and *TD* as the target data type.

ii) No other data type *Q* that is included as the source data type in the user-defined cast descriptor of some user-defined cast CF_Q that has *TD* as the target data type precedes *P* in the type precedence list of *SD*.

13) If *SD* is a collection type, then:

a) Let *ESD* be the element type of *SD*.

b) Let *ETD* be the element type of *TD*.

c) `CAST ( VALUE AS ETD )`

where *VALUE* is a <value expression> of declared type *ESD*, shall be a valid <cast specification>.

14) If *SD* is a row type, then:

a) Let *DSD* be the degree of *SD*.

b) Let *DTD* be the degree of *TD*.

c) *DSD* shall be equal to *DTD*.

d) Let FSD_i and FTD_i, 1 (one) $\le i \le$ *DSD*, be the *i*-th field of *SD* and *TD*, respectively.

e) Let $TFSD_i$ and $TFTD_i$, 1 (one) $\le i \le$ *DSD*, be the declared type of FSD_i and the declared type of FTD_i, respectively.

f) For *i* varying from 1 (one) to *DSD*, the <cast specification>:

`CAST ( VALUE`$_i$ `AS TFTD`$_i$ `)`

where $VALUE_i$ is an arbitrary <value expression> of declared type $TFSD_i$, shall be a valid <cast specification>.

15) If <domain name> is specified, then let *D* be the domain identified by the <domain name>. The schema identified by the explicit or implicit qualifier of the <domain name> shall include the descriptor of *D*.

Access Rules

1) If <domain name> is specified, then

Case:

a) If <cast specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the

<authorization identifier> that owns the containing schema shall include USAGE on the domain identified by <domain name>.

b) Otherwise, the current privileges shall include USAGE on the domain identified by <domain name>.

NOTE 102 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

2) If the <cast operand> is a <value expression> and either *SD* or *TD* is a user-defined type, then

Case:

a) If <cast specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include EXECUTE on CF_P.

b) Otherwise, the current privileges shall include EXECUTE on CF_P.

NOTE 103 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

General Rules

1) If the <cast operand> is a <value expression> *VE*, then let *SV* be its value.

2) Case:

a) If the <cast operand> specifies NULL, then *TV* is the null value and no further General Rules of this Subclause are applied.

b) If the <cast operand> specifies an <empty specification>, then *TV* is an empty collection of declared type *TD* and no further General Rules of this Subclause are applied.

c) If *SV* is the null value, then *TV* is the null value and no further General Rules of this Subclause are applied.

d) Otherwise, let *TV* be the result of the <cast specification> as specified in the remaining General Rules of this Subclause.

3) If either *SD* or *TD* is a user-defined type, then

Case:

a) If *TD* is a supertype of *SD*, then *TV* is *TR*.

b) Otherwise:

i) Let *CP* be the cast function contained in the user-defined cast descriptor of CF_P.

ii) The General Rules of Subclause 10.4, "<routine invocation>", are applied with a static SQL argument list that has a single SQL-argument that is <value expression> and with subject routine *CP*, yielding value *TR* that is the result of the invocation of *CP*.

iii) Case:

1) If *TD* is a user-defined type, then *TV* is *TR*.

2) Otherwise, *TV* is the result of

```
CAST (TR AS TD)
```

4) If either *SD* or *TD* is a reference type, then

   Case:

   a) If *RSTD* and *RTTD* are compatible, then:

      i) *TV* is *SV*.

      ii) The scope in the reference type descriptor of *TV* is *STD*.

   b) Otherwise:

      i) Let *CP* be the cast function contained in the user-defined cast descriptor of CF_P.

      ii) The General Rules of Subclause 10.4, "<routine invocation>", are applied with a static argument list that has a single SQL-argument that is a <value expression> and with subject routine *CP*, yielding value *TV* that is the result of the invocation of *CP*.

      iii) The scope in the reference type descriptor of *TV* is *STD*.

5) If *SD* is an array type, then:

   a) Let *SC* be the cardinality of *SV*.

   b) Let SVE_i be the *i*-th element of *SV*.

   c) For *i* varying from 1 (one) to *SC*, the following <cast specification> is applied:

      ```
      CAST ( SVE_i AS ETD )
      ```

      yielding value TVE_i.

   d) If *TD* is an array type, then let *TC* be the maximum cardinality of *TD*.

      Case:

      i) If *SC* is greater than *TC*, then an exception condition is raised: *data exception — array data, right truncation.*

      ii) Otherwise, *TV* is the array with elements TVE_i, 1 (one) $\leq i \leq$ *SC*.

   e) If *TD* is a multiset type, then *TV* is the multiset with elements TVE_i, 1 (one) $\leq i \leq$ *SC*.

6) If *SD* is a multiset type, then:

   a) Let *SC* be the cardinality of *SV*.

   b) The elements of *SV* are placed in an implementation-dependent order. Let SVE_i, 1 (one) $\leq i \leq$ *SC*, be the *i*-th element of *SV* in this ordering.

   c) For *i* varying from 1 (one) to *SC*, the following <cast specification> is applied:

      ```
      CAST ( SVE_i AS ETD )
      ```

yielding value TVE_i.

d) If *TD* is an array type, then let *TC* be the maximum cardinality of *TD*.

Case:

i) If *SC* is greater than *TC*, then an exception condition is raised: *data exception — array data, right truncation.*

ii) Otherwise, *TV* is the array with elements TVE_i, 1 (one) ≤ *i* ≤ *SC*.

e) If *TD* is a multiset type, then *TV* is the multiset with elements TVE_i, 1 (one) ≤ *i* ≤ *SC*.

7) If *SD* is a row type, then:

a) For *i* varying from 1 (one) to *DSD*, the <cast specification> is applied:

```
CAST ( FSDi AS TFTDi )
```

yielding a value TVE_i.

b) *TV* is `ROW ( TVE1, TVE2, ..., TVEDSD )`.

8) If *TD* is exact numeric, then

Case:

a) If *SD* is exact numeric or approximate numeric, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits after rounding or truncating if necessary, then *TV* is that representation. The choice of whether to round or truncate is implementation-defined.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range.*

b) If *SD* is character string, then *SV* is replaced by *SV* with any leading or trailing <space>s removed.

Case:

i) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 5.3, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast.*

ii) Otherwise, let *LT* be that <signed numeric literal>. The <cast specification> is equivalent to

```
CAST ( LT AS TD )
```

c) If *SD* is an interval data type, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits, then *TV* is that representation.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range.*

9) If *TD* is approximate numeric, then

Case:

a) If *SD* is exact numeric or approximate numeric, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits after rounding or truncating if necessary, then *TV* is that representation. The choice of whether to round or truncate is implementation-defined.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range.*

b) If *SD* is character string, then *SV* is replaced by *SV* with any leading or trailing <space>s removed.

Case:

i) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 5.3, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast.*

ii) Otherwise, let *LT* be that <signed numeric literal>. The <cast specification> is equivalent to

```
CAST ( LT AS TD )
```

10) If *TD* is fixed-length character string, then let *LTD* be the length in characters of *TD*.

Case:

a) If *SD* is exact numeric, then:

i) Let *YP* be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 5.3, "<literal>", whose scale is the same as the scale of *SD* and whose interpreted value is the absolute value of *SV*.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast.*

2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD–LY* <space>s.

4) Otherwise, an exception condition is raised: *data exception — string data, right truncation.*

b) If *SD* is approximate numeric, then:

i) Let *YP* be a character string as follows:

Case:

1) If *SV* equals 0 (zero), then *YP* is '0E0'.

2) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 5.3, "<literal>", whose interpreted value is equal to the absolute value of *SV* and whose <mantissa> consists of a single <digit> that is not '0' (zero), followed by a <period> and an <unsigned integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast.*

2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD–LY* <space>s.

4) Otherwise, an exception condition is raised: *data exception — string data, right truncation.*

c) If *SD* is fixed-length character string, variable-length character string, or large object character string, then

Case:

i) If the length in characters of *SV* is equal to *LTD*, then *TV* is *SV*.

ii) If the length in characters of *SV* is larger than *LTD*, then *TV* is the first *LTD* characters of *SV*. If any of the remaining characters of *SV* are non-<space> characters, then a completion condition is raised: *warning — string data, right truncation.*

iii) If the length in characters *M* of *SV* is smaller than *LTD*, then *TV* is *SV* extended on the right by *LTD–M* <space>s.

d) If *SD* is a datetime data type or an interval data type, then let *Y* be the shortest character string that conforms to the definition of <literal> in Subclause 5.3, "<literal>", and such that the interpreted value of *Y* is *SV* and the interpreted precision of *Y* is the precision of *SD*. If *SV* is an interval, then <sign> shall be specified within <unquoted interval string> in the literal *Y*.

Case:

i) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast.*

ii) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

iii) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD–LY* <space>s.

iv) Otherwise, an exception condition is raised: *data exception — string data, right truncation.*

e) If *SD* is boolean, then

Case:

i) If *SV* is *True* and *LTD* is not less than 4, then *TV* is 'TRUE' extended on the right by *LTD*–4 <space>s.

ii) If *SV* is *False* and *LTD* is not less than 5, then *TV* is 'FALSE' extended on the right by *LTD*–5 <space>s.

iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast.*

11) If *TD* is variable-length character string or large object character string, then let *MLTD* be the maximum length in characters of *TD*.

Case:

a) If *SD* is exact numeric, then:

i) Let *YP* be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 5.3, "<literal>", whose scale is the same as the scale of *SD* and whose interpreted value is the absolute value of *SV*.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast.*

2) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation.*

b) If *SD* is approximate numeric, then

i) Let *YP* be a character string as follows:

Case:

1) If *SV* equals 0 (zero), then *YP* is '0E0'.

2) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 5.3, "<literal>", whose interpreted value is equal to the absolute value of *SV* and whose <mantissa> consists of a single <digit> that is not '0', followed by a <period> and an <unsigned integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast.*

2) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation.*

c) If *SD* is fixed-length character string, variable-length character string, or large object character string, then

Case:

i) If the length in characters of *SV* is less than or equal to *MLTD*, then *TV* is *SV*.

ii) If the length in characters of *SV* is larger than *MLTD*, then *TV* is the first *MLTD* characters of *SV*. If any of the remaining characters of *SV* are non-<space> characters, then a completion condition is raised: *warning — string data, right truncation.*

d) If *SD* is a datetime data type or an interval data type then let *Y* be the shortest character string that conforms to the definition of <literal> in Subclause 5.3, "<literal>", and such that the interpreted value of *Y* is *SV* and the interpreted precision of *Y* is the precision of *SD*. If *SV* is a negative interval, then <sign> shall be specified within <unquoted interval string> in the literal *Y*.

Case:

i) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast.*

ii) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

iii) Otherwise, an exception condition is raised: *data exception — string data, right truncation.*

e) If *SD* is boolean, then

Case:

i) If *SV* is *True* and *MLTD* is not less than 4, then *TV* is 'TRUE'.

ii) If *SV* is *False* and *MLTD* is not less than 5, then *TV* is 'FALSE'.

iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast.*

12) If *TD* and *SD* are binary string data types, then let *MLTD* be the maximum length in octets of *TD*.

Case:

a) If the length in octets of *SV* is less than or equal to *MLTD*, then *TV* is *SV*.

b) If the length in octets of *SV* is larger than *MLTD*, then *TV* is the first *MLTD* octets of *SV* and a completion condition is raised: *warning — string data, right truncation.*

13) If *TD* is the datetime data type DATE, then

Case:

a) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted date string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format.*

iii) Otherwise, an exception condition is raised: *data exception — invalid datetime format.*

b) If *SD* is a date, then *TV* is *SV*.

c) If *SD* is the datetime data type TIMESTAMP WITHOUT TIME ZONE, then *TV* is the year, month, and day <primary datetime field>s of *SV*.

d) If *SD* is the datetime data type TIMESTAMP WITH TIME ZONE, then *TV* is computed by:

```
CAST ( CAST ( SV AS TIMESTAMP WITHOUT TIME ZONE ) AS DATE )
```

14) Let *STZD* be the current default time zone displacement of the SQL-session.

15) If *TD* is the datetime data type TIME WITHOUT TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITH TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

```
CAST ( TV1 AS TIME(TSP) WITHOUT TIME ZONE )
```

iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format.*

iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast.*

b) If *SD* is TIME WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.

c) If *SD* is TIME WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, computed modulo 24 hours, with implementation-defined rounding or truncation if necessary.

d) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined rounding or truncation if necessary.

e) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is:

```
CAST ( CAST ( SV AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
       AS TIME(TSP) WITHOUT TIME ZONE )
```

16) If *TD* is the datetime data type TIME WITH TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITHOUT TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

```
CAST ( TV1 AS TIME(TSP) WITH TIME ZONE )
```

iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format.*

iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast.*

b) If *SD* is TIME WITH TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.

c) If *SD* is TIME WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, computed modulo 24 hours, with implementation-defined rounding or truncation if necessary, and the time zone component of *TV* is *STZD*.

d) If *SD* is TIMESTAMP WITH TIME ZONE, then the UTC component of *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined rounding or truncation if necessary, and the time zone component of *TV* is the time zone displacement of *SV*.

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is:

```
CAST ( CAST ( SV AS TIMESTAMP(TSP) WITH TIME ZONE )
AS TIME(TSP) WITH TIME ZONE )
```

17) If *TD* is the datetime data type TIMESTAMP WITHOUT TIME ZONE, then let *TSP* be the <timestamp precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITH TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

```
CAST ( TV1 AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
```

iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format.*

iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast.*

b) If *SD* is a date, then the <primary datetime field>s hour, minute, and second of *TV* are set to 0 (zero) and the <primary datetime field>s year, month, and day of *TV* are set to their respective values in *SV*.

c) If *SD* is TIME WITHOUT TIME ZONE, then the <primary datetime field>s year, month, and day of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and second of *TV* are set to their respective values in *SV*, with implementation-defined rounding or truncation if necessary.

d) If *SD* is TIME WITH TIME ZONE, then *TV* is:

```
CAST ( CAST ( SV AS TIMESTAMP WITH TIME ZONE )
AS TIMESTAMP WITHOUT TIME ZONE )
```

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.

f) If *SD* is TIMESTAMP WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, with implementation-defined rounding or truncation if necessary.

18) If *TD* is the datetime data type TIMESTAMP WITH TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITHOUT TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

```
CAST ( TV1 AS TIMESTAMP(TSP) WITH TIME ZONE )
```

iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format.*

iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast.*

b) If *SD* is a date, then *TV* is:

```
CAST ( CAST ( SV AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
       AS TIMESTAMP(TSP) WITH TIME ZONE )
```

c) If *SD* is TIME WITHOUT TIME ZONE, then *TC* is:

```
CAST ( CAST ( SV AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
       AS TIMESTAMP(TSP) WITH TIME ZONE )
```

d) If *SD* is TIME WITH TIME ZONE, then the <primary datetime field>s of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and second are set to their respective values in *SV*, with implementation-defined rounding or truncation if necessary. The time zone component of *TV* is set to the time zone component of *SV*.

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, with a time zone displacement of *STZD*.

f) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is *SV* with implementation-defined rounding or truncation, if necessary.

19) If *TD* is interval, then

Case:

a) If *SD* is exact numeric, then

Case:

i) If the representation of *SV* in the data type *TD* would result in the loss of leading significant digits, then an exception condition is raised: *data exception — interval field overflow.*

ii) Otherwise, *TV* is that representation.

b) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted interval string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) Otherwise,

Case:

1) If a <datetime value> does not conform to the natural rules for intervals according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid interval format.*

2) Otherwise, an exception condition is raised: *data exception — invalid datetime format.*

c) If *SD* is interval and *TD* and *SD* have the same interval precision, then *TV* is *SV*.

d) If *SD* is interval and *TD* and *SD* have different interval precisions, then let *Q* be the least significant <primary datetime field> of *TD*.

i) Let *Y* be the result of converting *SV* to a scalar in units *Q* according to the natural rules for intervals as defined in the Gregorian calendar (that is, there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year).

ii) Normalize *Y* to conform to the <interval qualifier> "*P* TO *Q*" of *TD* (again, observing the rules that there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year). Whether to truncate or round in the least significant field of the result is implementation-defined. If this would result in loss of precision of the leading datetime field of *Y*, then an exception condition is raised: *data exception — interval field overflow.*

iii) *TV* is the value of *Y*.

20) If *TD* is boolean, then

Case:

a) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast.*

b) If *SD* is boolean, then *TV* is *SV*.

21) If the <cast specification> contains a <domain name> and that <domain name> refers to a domain that contains a <domain constraint> and if *TV* does not satisfy the <check constraint definition> simply contained in the <domain constraint>, then an exception condition is raised: *integrity constraint violation.*

Conformance Rules

1) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain a <cast operand> whose declared type is BINARY LARGE OBJECT or CHARACTER LARGE OBJECT.

2) Without Feature F421, "National character", conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER LARGE OBJECT.

3) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER LARGE OBJECT.

4) Without Feature S043, "Enhanced reference types", in conforming SQL language, if the declared data type of <cast operand> is a reference type, then <cast target> shall contain a <data type> that is a reference type.

6.13 <next value expression>

Function

Return the next value of a sequence generator.

Format

```
<next value expression> ::= NEXT VALUE FOR <sequence generator name>
```

Syntax Rules

1) A <next value expression> shall be directly contained in one of the following:

   a) A <select list> simply contained in a <query specification> that constitutes a <query expression> that is immediately contained in one of the following:

      i) A <cursor specification>.

      ii) A <subquery> simply contained in an <as subquery clause> in a <table definition>.

      iii) A <from subquery>.

      iv) A <select statement: single row>.

   b) A <select list> simply contained in a <query specification> that is immediately contained in a <dynamic single row select statement>.

   c) A <from constructor>.

   d) A <merge insert value list>.

   e) An <update source>.

2) <next value expression> shall not be contained in a <case expression>, a <search condition>, an <order by clause>, an <aggregate function>, a <window function>, a grouped query, or in a <query specification> that simply contains the <set quantifier> DISTINCT.

Access Rules

1) Case:

   a) If <next value expression> is contained in a <schema definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include USAGE privilege on the sequence generator identified by <sequence generator name>.

   b) Otherwise, the current privileges shall include USAGE privilege on the sequence generator identified by <sequence generator name>.

   NOTE 104 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

General Rules

1) If <next value expression> *NVE* is specified, then let *SEQ* be the sequence generator descriptor identified by the <sequence generator name> contained in *NVE*.

   Case:

   a) If *NVE* is directly contained in a <query specification> *QS*, then the General Rules of Subclause 9.21, "Generation of the next value of a sequence generator", are applied once per row in the result of *QS* with *SEQ* as *SEQUENCE*. The result of each evaluation of *NVE* for a given row is the *RESULT* returned by the General Rules of Subclause 9.21, "Generation of the next value of a sequence generator".

   b) If *NVE* is directly contained in a <contextually typed table value constructor> *TVC*, then the General Rules of Subclause 9.21, "Generation of the next value of a sequence generator", are applied once per <contextually typed row value expression> contained in *TVC*. The result of each evaluation of *NVE* for a given <row value expression> is the *RESULT* returned by the General Rules of Subclause 9.21, "Generation of the next value of a sequence generator".

   c) If *NVE* is directly contained in an <update source>, then the General Rules of Subclause 9.21, "Generation of the next value of a sequence generator", are applied once per row to be updated by the <update statement: searched> or <update statement: positioned>. The result of each evaluation of *NVE* for a given row is the *RESULT* returned by the General Rules of Subclause 9.21, "Generation of the next value of a sequence generator".

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <next value expression>.

6.14 <field reference>

Function

Reference a field of a row value.

Format

```
<field reference> ::= <value expression primary> <period> <field name>
```

Syntax Rules

1) Let *FR* be the <field reference>, let *VEP* be the <value expression primary> immediately contained in *FR*, and let *FN* be the <field name> immediately contained in *FR*.

2) The declared type of *VEP* shall be a row type. Let *RT* be that row type.

3) *FR* is a *field reference*.

4) *FN* shall unambiguously reference a field of *RT*. Let *F* be that field.

5) The declared type of *FR* is the declared type of *F*.

Access Rules

None.

General Rules

1) Let *VR* be the value of *VEP*.

2) Case:

   a) If *VR* is the null value, then the value of *FR* is the null value.

   b) Otherwise, the value of *FR* is the value of the field *F* of *VR*.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain a <field reference>.

6.15 <subtype treatment>

Function

Modify the declared type of an expression.

Format

```
<subtype treatment> ::=
    TREAT <left paren> <subtype operand> AS <target subtype> <right paren>

<subtype operand> ::= <value expression>

<target subtype> ::=
    <path-resolved user-defined type name>
  | <reference type>
```

Syntax Rules

1) The declared type *VT* of the <value expression> shall be a structured type or a reference type.

2) Case:

   a) If *VT* is a structured type, then:

      i) <target subtype> shall specify a <path-resolved user-defined type name>.

      ii) Let *DT* be the structured type identified by the <user-defined type name> simply contained in <path-resolved user-defined type name>.

   b) Otherwise:

      i) <target subtype> shall specify a <reference type>.

      ii) Let *DT* be the reference type identified by <reference type>.

3) *VT* shall be a supertype of *DT*.

4) The declared type of the result of the <subtype treatment> is *DT*.

Access Rules

None.

General Rules

1) Let *V* be the value of the <value expression>.

2) Case:

   a) If *V* is the null value, then the value of the <subtype treatment> is the null value.

b) Otherwise:

i) If the most specific type of *V* is not a subtype of *DT*, then an exception condition is raised: *invalid target type specification.*

NOTE 105 — "most specific type" is defined in Subclause 4.7.5, "Subtypes and supertypes".

ii) The value of the <subtype treatment> is *V*.

Conformance Rules

1) Without Feature S161, "Subtype treatment", conforming SQL Language shall not contain a <subtype treatment>.

2) Without Feature S162, "Subtype treatment for references", conforming SQL language shall not contain a <target subtype> that contains a <reference type>.

6.16 <method invocation>

Function

Reference an SQL-invoked method of a user-defined type value.

Format

```
<method invocation> ::=
    <direct invocation>
  | <generalized invocation>

<direct invocation> ::=
    <value expression primary> <period> <method name> [ <SQL argument list> ]

<generalized invocation> ::=
    <left paren> <value expression primary> AS <data type> <right paren>
    <period> <method name> [ <SQL argument list> ]

<method selection> ::= <routine invocation>

<constructor method selection> ::= <routine invocation>
```

Syntax Rules

1) Let *OR* be the <method invocation>, let *VEP* be the <value expression primary> immediately contained in the <direct invocation> or <generalized invocation> of *OR*, and let *MN* be the <method name> immediately contained in *OR*.

2) The declared type of *VEP* shall be a user-defined type. Let *UDT* be that user-defined type.

3) Case:

   a) If <SQL argument list> is specified, then let *AL* be:

      $$, A_1, \ldots, A_n$$

      where A_i, 1 (one) $\leq i \leq n$, are the <SQL argument>s immediately contained in <SQL argument list>, taken in order of their ordinal position in <SQL argument list>.

   b) Otherwise, let *AL* be a zero-length string.

4) Case:

   a) If <method invocation> is immediately contained in <new invocation>, then let *TP* be an SQL-path containing the <schema name> of the schema that includes the descriptor of *UDT*.

   b) Otherwise, let *TP* be an SQL-path, arbitrarily defined, containing the <schema name> of every schema that includes a descriptor of a supertype or subtype of *UDT*.

5) Case:

a) If <generalized invocation> is specified, then let *DT* be the <data type> simply contained in the <generalized invocation>. Let *RI* be the following <method selection>:

```
MN (VEP AS DT AL)
```

b) Otherwise,

Case:

i) If <method invocation> is immediately contained in <new invocation>, then let *RI* be the <constructor method selection>:

```
MN (VEP AL)
```

ii) Otherwise, let *RI* be the following <method selection>:

```
MN (VEP AL)
```

6) The Syntax Rules of Subclause 10.4, "<routine invocation>", are applied with *RI* and *TP* as the <routine invocation> and SQL-path, respectively, yielding subject routine *SR* and static SQL argument list *SAL*.

Access Rules

None.

General Rules

1) The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SR* and *SAL* as the subject routine and SQL argument list, respectively, yielding value *V* that is the result of the <routine invocation>.

2) The value of <method invocation> is *V*.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <method invocation>.

6.17 <static method invocation>

Function

Invoke a static method.

Format

```
<static method invocation> ::=
    <path-resolved user-defined type name> <double colon> <method name>
    [ <SQL argument list> ]

<static method selection> ::= <routine invocation>
```

Syntax Rules

1) Let *TN* be the <user-defined type name> immediately contained in <path-resolved user-defined type name> and let *T* be the user-defined type identified by *TN*.

2) Let *MN* be the <method name> immediately contained in <static method invocation>.

3) Case:

   a) If <SQL argument list> is specified, then let *AL* be that <SQL argument list>.

   b) Otherwise, let *AL* be <left paren> <right paren>.

4) Let *TP* be an SQL-path containing only the <schema name> of every schema that includes a descriptor of a supertype of *T*.

5) Let *RI* be the following <routine invocation>:

   ```
   MN AL
   ```

6) Let *SMS* be the following <static method selection>:

   ```
   RI
   ```

7) The Syntax Rules of Subclause 10.4, "<routine invocation>", are applied with *RI* as the <routine invocation> immediately contained in the <static method selection> *SMS*, with *TP* as the SQL-path, and with *T* as the user-defined type of the static SQL-invoked method, yielding subject routine *SR* and static SQL argument list *SAL*.

Access Rules

None.

General Rules

1) The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SR* and *SAL* as the subject routine and SQL argument list, respectively, yielding a value *V* that is the result of the <routine invocation>.

2) The value of <static method invocation> is *V*.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <static method invocation>.

6.18 <new specification>

Function

Invoke a method on a newly-constructed value of a structured type.

Format

```
<new specification> ::=
    NEW <path-resolved user-defined type name> <SQL argument list>

<new invocation> ::=
    <method invocation>
  | <routine invocation>
```

Syntax Rules

1) Let *UDTN* be the <path-resolved user-defined type name> immediately contained in the <new specification>. Let *MN* be the <qualified identifier> immediately contained in *UDTN*.

2) Let *UDT* be the user-defined type identified by *UDTN*. *UDT* shall be instantiable. Let *SN* be the implicit or explicit <schema name> of *UDTN*. Let *S* be the schema identified by *SN*. Let *RN* be *NS.MN*.

3) Case:

   a) If the <new specification> is of the form

      NEW *UDTN*()

      then

      Case:

      i) If *S* does not include the descriptor of an SQL-invoked constructor method whose method name is equivalent to *MN* and whose unaugmented parameter list is empty, then the <new specification> is equivalent to the <new invocation>

         RN()

      ii) Otherwise, the <new specification> is equivalent to the <new invocation>

         RN().*MN*()

   b) Otherwise, the <new specification>

      NEW *UDTN*(a_1, a_2, ..., a_n)

      is equivalent to the <new invocation>

      RN().*MN*(a_1, a_2, ..., a_n)

Access Rules

None.

NOTE 106 — The applicable privileges or current privileges (as appropriate) include EXECUTE privilege on the constructor function, and also on the indicated constructor method, according to the Syntax Rules of Subclause 10.4, "<routine invocation>".

General Rules

None.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <new specification>.

6.19 <attribute or method reference>

Function

Return a value acquired by accessing a column of the row identified by a value of a reference type or by invoking an SQL-invoked method.

Format

```
<attribute or method reference> ::=
    <value expression primary> <dereference operator> <qualified identifier>
    [ <SQL argument list> ]

<dereference operator> ::= <right arrow>
```

Syntax Rules

1) The declared type of the <value expression primary> *VEP* shall be a reference type and the scope included in its reference type descriptor shall not be empty. Let *RT* be the referenced type of *VEP*.

2) Let *QI* be the <qualified identifier>. If <SQL argument list> is specified, then let *SAL* be <SQL argument list>; otherwise, let *SAL* be a zero-length string.

3) Case:

   a) If *QI* is equivalent to the attribute name of an attribute of *RT* and *SAL* is a zero-length string, then <attribute or method reference> is effectively replaced by a <dereference operation> *AMR* of the form:

      ```
      VEP -> QI
      ```

   b) Otherwise, <attribute or method reference> is effectively replaced by a <method reference> *AMR* of the form:

      ```
      VEP -> QI SAL
      ```

4) The declared type of <attribute or method reference> is the declared type of *AMR*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S041, "Basic reference types", conforming SQL language shall not contain an <attribute or method reference>.

6.20 <dereference operation>

Function

Access a column of the row identified by a value of a reference type.

Format

```
<dereference operation> ::=
    <reference value expression> <dereference operator> <attribute name>
```

Syntax Rules

1) Let *RVE* be the <reference value expression>. The reference type descriptor of *RVE* shall include a scope. Let *RT* be the referenced type of *RVE*.

2) Let *AN* be the <attribute name>. *AN* shall identify an attribute *AT* of *RT*.

3) The declared type of the <dereference operation> is the declared type of *AT*.

4) Let *S* be the name of the referenceable table in the scope of the reference type of *RVE*.

5) Let *OID* be the name of the self-referencing column of *S*.

6) <dereference operation> is equivalent to a <scalar subquery> of the form:

```
( SELECT AN
  FROM S
  WHERE S.OID = RVE )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <dereference operation>.

6.21 <method reference>

Function

Return a value acquired from invoking an SQL-invoked routine that is a method.

Format

```
<method reference> ::=
    <value expression primary> <dereference operator> <method name> <SQL argument list>
```

Syntax Rules

1) The data type of the <value expression primary> *VEP* shall be a reference type and the scope included in its reference type descriptor shall not be empty.

2) Let *MN* be the method name. Let *MRAL* be the <SQL argument list>.

3) The Syntax Rules of Subclause 6.16, "<method invocation>", are applied to the <method invocation>:

   ```
   DEREF (VEP) . MN MRAL
   ```

   yielding subject routine *SR* and static SQL argument list *SAL*.

4) The data type of <method reference> is the data type of the expression:

   ```
   DEREF (VEP) . MN MRAL
   ```

Access Rules

1) Let *SCOPE* be the table that is the scope of *VEP*.

   Case:

   a) If <method reference> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include the table/method privilege for table *SCOPE* and method *SR*.

   b) Otherwise, the current privileges shall include the table/method privilege for table *SCOPE* and method *SR*.

General Rules

1) The General Rules of Subclause 6.16, "<method invocation>", are applied with *SR* and *SAL* as the subject routine and SQL argument list, respectively, yielding a value *V* that is the result of the <routine invocation>.

2) The value of <method reference> is *V*.

Conformance Rules

1) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <method reference>.

6.22 <reference resolution>

Function

Obtain the value referenced by a reference value.

Format

```
<reference resolution> ::=
    DEREF <left paren> <reference value expression> <right paren>
```

Syntax Rules

1) Let *RR* be the <reference resolution> and let *RVE* be the <reference value expression>. The reference type descriptor of *RVE* shall include a scope.

2) The declared type of *RR* is the structured type that is referenced by the declared type of *RVE*.

3) Let *SCOPE* be the table identified by the table name included in the reference type descriptor of *RVE*. *SCOPE* is the scoped table of *RR*.

   NOTE 107 — The term "scoped table" is defined in Subclause 4.9, "Reference types".

4) Let m be the number of subtables of *SCOPE*. Let S_i, 1 (one) $\leq i \leq m$, be the subtables, arbitrarily ordered, of *SCOPE*.

5) For each S_i, 1 (one) $\leq i \leq m$, let STN_i be the name included in the descriptor of S_i of the structured type ST_i associated with S_i, let $REFCOL_i$ be the self-referencing column of S_i, let N_i be the number of attributes of ST_i, and let $A_{i,j}$, 1 (one) $\leq j \leq N_i$, be the names of the attributes of ST_i, therefore also the names of the columns of S_i.

Access Rules

1) Case:

   a) If <reference resolution> is contained in a <schema definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *SCOPE*.

   b) Otherwise, the current privileges shall includes SELECT WITH HIERARCHY OPTION on at least one supertable of *SCOPE*.

   NOTE 108 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

General Rules

1) The value of <reference resolution> is the value of:

```
(
  SELECT A1,1 ( ... A1,N1
         ( STN1(), A1,N1 ), ... A1,1 )
  FROM ONLY S1
  WHERE S1.REFCOL1 = RVE
 UNION
  SELECT A2,1 ( ... A2,N2
         ( STN2(), A2,N2 ), ... A2,1 )
  FROM ONLY S2
  WHERE S2.REFCOL2 = RVE
 UNION
 ...
 UNION
  SELECT Am,1 ( ... Am,Nm
         ( STNm(), Am,Nm ), ... Am,1 )
  FROM ONLY Sm
  WHERE Sm.REFCOLm = RVE
)
```

NOTE 109 — The evaluation of this General Rule is effectively performed without further Access Rule checking.

Conformance Rules

1) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <reference resolution>.

6.23 <array element reference>

Function

Return an element of an array.

Format

```
<array element reference> ::=
    <array value expression>
    <left bracket or trigraph> <numeric value expression> <right bracket or trigraph>
```

Syntax Rules

1) The declared type of an <array element reference> is the element type of the specified <array value expression>.

2) The declared type of <numeric value expression> shall be exact numeric with scale 0 (zero).

Access Rules

None.

General Rules

1) If the value of <array value expression> or <numeric value expression> is the null value, then the result of <array element reference> is the null value.

2) Let the value of <numeric value expression> be *i*.

   Case:

   a) If *i* is greater than zero and less than or equal to the cardinality of <array value expression>, then the result of <array element reference> is the value of the *i*-th element of the value of <array value expression>.

   b) Otherwise, an exception condition is raised: *data exception — array element error*.

Conformance Rules

1) Without Feature S091, "Basic array support", conforming SQL language shall not contain an <array element reference>.

6.24 <multiset element reference>

Function

Return the sole element of a multiset of one element.

Format

```
<multiset element reference> ::=
    ELEMENT <left paren> <multiset value expression> <right paren>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression>. The <multiset element reference> is equivalent to the <scalar subquery>

```
( SELECT M.E
  FROM UNNEST (MSE) AS M(E) )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset element reference>.

6.25 <value expression>

Function

Specify a value.

Format

```
<value expression> ::=
    <common value expression>
  | <boolean value expression>
  | <row value expression>

<common value expression> ::=
    <numeric value expression>
  | <string value expression>
  | <datetime value expression>
  | <interval value expression>
  | <user-defined type value expression>
  | <reference value expression>
  | <collection value expression>

<user-defined type value expression> ::= <value expression primary>

<reference value expression> ::= <value expression primary>

<collection value expression> ::=
    <array value expression>
  | <multiset value expression>
```

Syntax Rules

1) The declared type of a <value expression> is the declared type of the simply contained <common value expression>, <boolean value expression>, or <row value expression>.

2) The declared type of a <common value expression> is the declared type of the <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <user-defined type value expression>, <collection value expression>, or <reference value expression>, respectively.

3) The declared type of a <user-defined type value expression> is the declared type of the immediately contained <value expression primary>, which shall be a user-defined type.

4) The declared type of a <reference value expression> is the declared type of the immediately contained <value expression primary>, which shall be a reference type.

5) The declared type of a <collection value expression> is the declared type of the immediately contained <array value expression> or <multiset value expression>.

6) Let *C* be some column. Let *VE* be the <value expression>. *C* is an underlying column of *VE* if and only if *C* is identified by some column reference contained in *VE*.

7) A <value expression> or <nonparenthesized value expression primary> is *possibly non-deterministic* if it generally contains any of the following:

   a) A <datetime value function>.

   b) A <next value expression>.

   c) A <cast specification> that either is, or recursively implies through the execution of the General Rules of Subclause 6.12, "<cast specification>", one of the following:

      i) A <cast specification> whose result type is datetime with time zone and whose <cast operand> has declared type that is not datetime with time zone.

      ii) A <cast specification> whose result type is an array type and whose <cast operand> has a declared type that is a multiset type.

   d) An <array value constructor by query>.

   e) A <datetime factor> that simply contains a <datetime primary> whose declared type is datetime without time zone and that simply contains an explicit <time zone>.

   f) An <interval value expression> that computes the difference of a <datetime value expression> and a <datetime term>, such that the declared type of one operand is datetime with time zone and the other operand is datetime without time zone.

   g) A <comparison predicate> , <overlaps predicate>, or <distinct predicate> simply containing <row value predicand>s *RVP1* and *RVP2* such that the declared types of *RVP1* and *RVP2* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

      NOTE 110 — This includes <between predicate> because of a syntactic transformation to <comparison predicate>.

   h) A <quantified comparison predicate> or a <match predicate> simply containing a <row value predicand> *RVP* and a <table subquery> *TS* such that the declared types of *RVP* and *TS* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

      NOTE 111 — This includes <in predicate> because of a syntactic transformation to <quantified comparison predicate>.

   i) A <member predicate> simply containing a <row value predicand> *RVP* and a <multiset value expression> *MVP* such that the declared type of the only field *F* of *RVP* and the element type of *MVP* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

   j) A <submultiset predicate> simply containing a <row value predicand> *RVP* and a <multiset value expression> *MVP* such that the declared type of the only field *F* of *RVP* and the declared type of *MVP* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

   k) A <multiset value expression> that specifies or implies MULTISET UNION, MULTISET EXCEPT, or MULTISET INTERSECT such that the element types of the operands have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

   l) A <value specification> that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, or CURRENT_PATH.

m) A <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.

n) An <aggregate function> that specifies MIN or MAX and that simply contains a <value expression> whose declared type is based on a character string type, user-defined type, or datetime with time zone type.

o) An <aggregate function> that specifies INTERSECTION and that simply contains a <value expression> whose declared element type is based on a character string type, a user-defined type, or a datetime type with time zone.

p) A <multiset value expression> that specifies MULTISET UNION DISTINCT, MULTISET EXCEPT, or MULTISET INTERSECT and whose result type's declared element type is based on character string type, a user-defined type, or a datetime type with time zone.

q) A <multiset set function> whose declared element type is based on a character string type, a user-defined type, or a datetime type with time zone.

r) A <window function> that specifies ROW_NUMBER or whose associated <window specification> specifies ROWS.

s) A <query specification> or <query expression> that is possibly non-deterministic.

Access Rules

None.

General Rules

1) The value of a <value expression> is the value of the simply contained <common value expression>, <boolean value expression>, or <row value expression>.

2) The value of a <common value expression> is the value of the immediately contained <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <user-defined type value expression>, <collection value expression>, or <reference value expression>.

3) When a <value expression> *V* is evaluated for a row *R* of a table, each reference to a column of that table by a column reference *CR* directly contained in *V* is the value of that column in that row.

4) The value of a <collection value expression> is the value of its immediately contained <array value expression> or <multiset value expression>.

5) The value of a <reference value expression> *RVE* is the value of the <value expression primary> immediately contained in *RVE*.

Conformance Rules

1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <value expression> that is a <boolean value expression>.

2) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <reference value expression>.

6.26 <numeric value expression>

Function

Specify a numeric value.

Format

```
<numeric value expression> ::=
    <term>
  | <numeric value expression> <plus sign> <term>
  | <numeric value expression> <minus sign> <term>

<term> ::=
    <factor>
  | <term> <asterisk> <factor>
  | <term> <solidus> <factor>

<factor> ::= [ <sign> ] <numeric primary>

<numeric primary> ::=
    <value expression primary>
  | <numeric value function>
```

Syntax Rules

1) If the declared type of both operands of a dyadic arithmetic operator is exact numeric, then the declared type of the result is an implementation-defined exact numeric type, with precision and scale determined as follows:

   a) Let *S1* and *S2* be the scale of the first and second operands respectively.

   b) The precision of the result of addition and subtraction is implementation-defined, and the scale is the maximum of *S1* and *S2*.

   c) The precision of the result of multiplication is implementation-defined, and the scale is *S1* + *S2*.

   d) The precision and scale of the result of division are implementation-defined.

2) If the declared type of either operand of a dyadic arithmetic operator is approximate numeric, then the declared type of the result is an implementation-defined approximate numeric type.

3) The declared type of a <factor> is that of the immediately contained <numeric primary>.

4) The declared type of a <numeric primary> shall be numeric.

5) If a <numeric value expression> immediately contains a <minus sign> *NMS* and immediately contains a <term> that immediately contains a <factor> that immediately contains a <sign> that is a <minus sign> *FMS*, then there shall be a <separator> between *NMS* and *FMS*.

Access Rules

None.

General Rules

1) If the value of any <numeric primary> simply contained in a <numeric value expression> is the null value, then the result of the <numeric value expression> is the null value.

2) If the <numeric value expression> contains only a <numeric primary>, then the result of the <numeric value expression> is the value of the specified <numeric primary>.

3) The monadic arithmetic operators <plus sign> and <minus sign> (+ and –, respectively) specify monadic plus and monadic minus, respectively. Monadic plus does not change its operand. Monadic minus reverses the sign of its operand.

4) The dyadic arithmetic operators <plus sign>, <minus sign>, <asterisk>, and <solidus> (+, –, *, and /, respectively) specify addition, subtraction, multiplication, and division, respectively. If the value of a divisor is zero, then an exception condition is raised: *data exception — division by zero.*

5) If the most specific type of the result of an arithmetic operation is exact numeric, then

   Case:

   a) If the operator is not division and the mathematical result of the operation is not exactly representable with the precision and scale of the result data type, then an exception condition is raised: *data exception — numeric value out of range.*

   b) If the operator is division and the approximate mathematical result of the operation represented with the precision and scale of the result data type loses one or more leading significant digits after rounding or truncating if necessary, then an exception condition is raised: *data exception — numeric value out of range.* The choice of whether to round or truncate is implementation-defined.

6) If the most specific type of the result of an arithmetic operation is approximate numeric and the exponent of the approximate mathematical result of the operation is not within the implementation-defined exponent range for the result data type, then an exception condition is raised: *data exception — numeric value out of range.*

Conformance Rules

None.

6.27 <numeric value function>

Function

Specify a function yielding a value of type numeric.

Format

```
<numeric value function> ::=
    <position expression>
  | <extract expression>
  | <length expression>
  | <cardinality expression>
  | <absolute value expression>
  | <modulus expression>
  | <natural logarithm>
  | <exponential function>
  | <power function>
  | <square root>
  | <floor function>
  | <ceiling function>
  | <width bucket function>

<position expression> ::=
    <string position expression>
  | <blob position expression>

<string position expression> ::=
    POSITION <left paren> <string value expression> IN <string value expression>
    [ USING <char length units> ] <right paren>

<blob position expression> ::=
    POSITION <left paren> <blob value expression> IN <blob value expression> <right paren>

<length expression> ::=
    <char length expression>
  | <octet length expression>

<char length expression> ::=
    { CHAR_LENGTH | CHARACTER_LENGTH } <left paren> <string value expression>
    [ USING <char length units> ] <right paren>

<octet length expression> ::=
    OCTET_LENGTH <left paren> <string value expression> <right paren>

<extract expression> ::=
    EXTRACT <left paren> <extract field> FROM <extract source> <right paren>

<extract field> ::=
    <primary datetime field>
  | <time zone field>

<time zone field> ::=
    TIMEZONE_HOUR
```

```
  | TIMEZONE_MINUTE

<extract source> ::=
    <datetime value expression>
  | <interval value expression>

<cardinality expression> ::=
    CARDINALITY <left paren> <collection value expression> <right paren>

<absolute value expression> ::= ABS <left paren> <numeric value expression> <right paren>

<modulus expression> ::=
    MOD <left paren> <numeric value expression dividend> <comma>
    <numeric value expression divisor><right paren>

<numeric value expression dividend> ::= <numeric value expression>

<numeric value expression divisor> ::= <numeric value expression>

<natural logarithm> ::= LN <left paren> <numeric value expression> <right paren>

<exponential function> ::= EXP <left paren> <numeric value expression> <right paren>

<power function> ::=
    POWER <left paren> <numeric value expression base> <comma>
    <numeric value expression exponent> <right paren>

<numeric value expression base> ::= <numeric value expression>

<numeric value expression exponent> ::= <numeric value expression>

<square root> ::= SQRT <left paren> <numeric value expression> <right paren>

<floor function> ::= FLOOR <left paren> <numeric value expression> <right paren>

<ceiling function> ::=
    { CEIL | CEILING } <left paren> <numeric value expression> <right paren>

<width bucket function> ::=
    WIDTH_BUCKET <left paren> <width bucket operand> <comma> <width bucket bound 1> <comma>
    <width bucket bound 2> <comma> <width bucket count> <right paren>

<width bucket operand> ::= <numeric value expression>

<width bucket bound 1> ::= <numeric value expression>

<width bucket bound 2> ::= <numeric value expression>

<width bucket count> ::= <numeric value expression>
```

Syntax Rules

1) If <position expression> is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

2) If <string position expression> is specified, then both <string value expression>s shall be <character value expression>s that are comparable.

3) Case:

   a) If the character encoding form of <string value expression> is not UTF8, UTF16, or UTF32, then <char length units> shall not be specified.

   b) Otherwise, if <char length units> is not specified, then CHARACTERS is implicit.

4) If <extract expression> is specified, then

   Case:

   a) If <extract field> is a <primary datetime field>, then it shall identify a <primary datetime field> of the <interval value expression> or <datetime value expression> immediately contained in <extract source>.

   b) If <extract field> is a <time zone field>, then the declared type of the <extract source> shall be TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE.

5) If <extract expression> is specified, then

   Case:

   a) If <primary datetime field> does not specify SECOND, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

   b) Otherwise, the declared type of the result is an implementation-defined exact numeric type with scale not less than the specified or implied <time fractional seconds precision> or <interval fractional seconds precision>, as appropriate, of the SECOND <primary datetime field> of the <extract source>.

6) If a <length expression> is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

7) If <cardinality expression> is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

8) If <absolute value expression> is specified, then the declared type of the result is the declared type of the immediately contained <numeric value expression>.

9) If <modulus expression> is specified, then the declared type of each <numeric value expression> shall be exact numeric with scale 0 (zero). The declared type of the result is the declared type of the immediately contained <numeric value expression divisor>.

10) The declared type of the result of <natural logarithm> is an implementation-defined approximate numeric type.

11) The declared type of the result of <exponential function> is an implementation-defined approximate numeric type.

12) The declared type of the result of <power function> is an implementation-defined approximate numeric type.

13) If <square root> is specified, then let *NVE* be the simply contained <numeric value expression>. The <square root> is equivalent to

```
POWER (NVE, 0.5)
```

14) If <floor function> or <ceiling function> is specified, then

Case:

a) If the declared type of the simply contained <numeric value expression> *NVE* is exact numeric, then the declared type of the result is exact numeric with implementation-defined precision, with the radix of *NVE*, and with scale 0 (zero).

b) Otherwise, the declared type of the result is approximate numeric with implementation-defined precision.

15) If <width bucket function> is specified, then the declared type of <width bucket count> shall be exact numeric with scale 0 (zero). The declared type of the result of <width bucket function> is the declared type of <width bucket count>.

Access Rules

None.

General Rules

1) If the value of one or more <string value expression>s, <datetime value expression>s, <interval value expression>s, and <collection value expression>s that are simply contained in a <numeric value function> is the null value, then the result of the <numeric value function> is the null value.

2) If <string position expression> is specified, then let *SVE1* be the value of the first <string value expression> and let *SVE2* be the value of the second <string value expression>.

Case:

a) If CHAR_LENGTH(*SVE1*) is 0 (zero), then the result is 1 (one).

b) If <char length units> is specified, then let *CLU* be <char length units>; otherwise, let *CLU* be CHARACTERS. If there is at least one value *P* such that

```
SVE1 = SUBSTRING ( SVE2 FROM P FOR CHAR_LENGTH (SVE1 USING CLU ) USING CLU )
```

then the result is the least such *P*.

NOTE 112 — The collation used is determined in the normal way.

c) Otherwise, the result is 0 (zero).

3) If <blob position expression> is specified, then

Case:

a) If the first <blob value expression> has a length of 0 (zero), then the result is 1 (one).

b) If the value of the first <blob value expression> is equal to an identical-length substring of contiguous octets from the value of the second <blob value expression>, then the result is 1 (one) greater than the number of octets within the value of the second <blob value expression> preceding the start of the first such substring.

c) Otherwise, the result is 0 (zero).

4) If <extract expression> is specified, then

Case:

a) If <extract field> is a <primary datetime field>, then the result is the value of the datetime field identified by that <primary datetime field> and has the same sign as the <extract source>.

NOTE 113 — If the value of the identified <primary datetime field> is zero or if <extract source> is not an <interval value expression>, then the sign is irrelevant.

b) Otherwise, let *TZ* be the interval value of the implicit or explicit time zone displacement associated with the <datetime value expression>.

Case:

i) If <extract field> is TIMEZONE_HOUR, then the result is calculated as `EXTRACT (HOUR FROM TZ)`.

ii) Otherwise, the result is calculated as `EXTRACT (MINUTE FROM TZ)`

5) If a <char length expression> is specified, then

Case:

a) If the character encoding form of <character value expression> is not UTF8, UTF16, or UTF32, then let *S* be the <string value expression>.

Case:

i) If the most specific type of *S* is character string, then the result is the number of characters in the value of *S*.

NOTE 114 — The number of characters in a character string is determined according to the semantics of the character set of that character string.

ii) Otherwise, the result is OCTET_LENGTH(*S*).

b) Otherwise, the result is the number of explicit or implicit <char length units> in <char length expression>, counted in accordance with the definition of those units in the relevant normatively referenced document.

6) If an <octet length expression> is specified, then let *S* be the <string value expression>. Let *BL* be the number of bits (binary digits) in the value of *S*. The result of the <octet length expression> is the smallest integer not less than the quotient of the division (*BL*/8).

7) The result of <cardinality expression> is the number of elements of the result of the <collection value expression>.

8) If <absolute value expression> is specified, then let *N* be the value of the immediately contained <numeric value expression>.

Case:

a) If *N* is the null value, then the result is the null value.

b) If $N \geq 0$, then the result is *N*.

c) Otherwise, the result is $-1 * N$. If $-1 * N$ is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range.*

9) If <modulus expression> is specified, then let *N* be the value of the immediately contained <numeric value expression dividend> and let *M* be the value of the immediately contained <numeric value expression divisor>.

Case:

a) If either *N* or *M* is the null value, then the result is the null value.

b) If *M* is zero, then an exception condition is raised: *data exception — division by zero*.

c) Otherwise, the result is the unique nonnegative exact numeric value *R* with scale 0 (zero) such that all of the following are true:

i) *R* has the same sign as *N*.

ii) The absolute value of *R* is less than the absolute value of *M*.

iii) $N = M * K + R$ for some exact numeric value *K* with scale 0 (zero).

10) If <natural logarithm> is specified, then let *V* be the value of the simply contained <numeric value expression>.

Case:

a) If *V* is the null value, then the result is the null value.

b) If *V* is 0 (zero) or negative, then an exception condition is raised: *data exception — invalid argument for natural logarithm*.

c) Otherwise, the result is the natural logarithm of *V*.

11) If <exponential function> is specified, then let *V* be the value of the simply contained <numeric value expression>.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise, the result is *e* (the base of natural logarithms) raised to the power *V*. If the result is not representable in the the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range*.

12) If <power function> is specified, then let *NVEB* be the <numeric value expression base>, then let *VB* be the value of *NVEB*, let *NVEE* be the <numeric value expression exponent>, and let *VE* be the value of *NVEE*.

Case:

a) If either *VB* or *VE* is the null value, then the result is the null value.

b) If *VB* is 0 (zero) and *VE* is negative, then an exception condition is raised: *data exception — invalid argument for power function*.

c) If *VB* is 0 (zero) and *VE* is 0 (zero), then the result is 1 (one).

d) If *VB* is 0 (zero) and *VE* is positive, then the result is 0 (zero).

e) If *VB* is negative and *VE* is not equal to an exact numeric value with scale 0 (zero), then an exception condition is raised: *data exception — invalid argument for power function.*

f) If *VB* is negative and *VE* is equal to an exact numeric value with scale 0 (zero) that is an even number, then the result is the result of

```
EXP(NVEE*LN(-NVEB))
```

g) If *VB* is negative and *VE* is equal to an exact numeric value with scale 0 (zero) that is an odd number, then the result is the result of

```
-EXP(NVEE*LN(-NVEB))
```

h) Otherwise, the result is the result of

```
EXP(NVEE*LN(NVEB))
```

13) If <floor function> is specified, then let *V* be the value of the simply contained <numeric value expression> *NVE*.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise,

Case:

i) If the most specific type of *NVE* is exact numeric, then the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to *V*. If this result is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range.*

ii) Otherwise, the result is the greatest whole number that is less than or equal to *V*. If this result is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range.*

14) If <ceiling function> is specified, then let *V* be the value of the simply contained <numeric value expression> *NVE*.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise,

Case:

i) If the most specific type of *NVE* is exact numeric, then the result is the least exact numeric value with scale 0 (zero) that is greater than or equal to *V*. If this result is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range.*

ii) Otherwise, the result is the least whole number that is greater than or equal to *V*. If this result is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range.*

15) If <width bucket function> is specified, then let *WBO* be the value of <width bucket operand>, let *WBB1* be the value of <width bucket bound 1>, let *WBB2* be the value of <width bucket bound 2>, and let *WBC* be the value of <width bucket count>.

Case:

a) If any of *WBO*, *WBB1*, *WBB2*, or *WBC* is the null value, then the result is the null value.

b) If *WBC* is less than or equal to 0 (zero), then an exception condition is raised: *data exception — invalid argument for width bucket function.*

c) If *WBB1* equals *WBB2*, then an exception condition is raised: *data exception — invalid argument for width bucket function.*

d) If *WBB1* is less than *WBB2*, then

Case:

i) If *WBO* is less than *WBB1*, then the result is 0 (zero).

ii) If *WBO* is greater than or equal to *WBB2*, then the result is *WBC*+1. If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range.*

iii) Otherwise, the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to `((WBC * (WBO - WBB1) / (WBB2 - WBB1)) + 1)`

e) If *WBB1* is greater than *WBB2*, then

Case:

i) If *WBO* is greater than *WBB1*, then the result is 0 (zero).

ii) If *WBO* is less than or equal to *WBB2*, then the result is *WBC*+1. If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range.*

iii) Otherwise, the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to `((WBC * (WBB1 - WBO) / (WBB1 - WBB2)) + 1)`

Conformance Rules

1) Without Feature S091, "Basic array support", or Feature S271, "Basic multiset support", conforming SQL language shall not contain a <cardinality expression>.

2) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <extract expression>.

3) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <extract expression> that specifies a <time zone field>.

4) Feature F411, "Time zone specification", conforming SQL language shall not contain an <extract expression> that specifies a <time zone field>.

5) Without Feature F421, "National character", conforming SQL language shall not contain a <length expression> that simply contains a <string value expression> that has a declared type of NATIONAL CHARACTER LARGE OBJECT.

6) Without Feature T441, "ABS and MOD functions", conforming language shall not contain an <absolute value expression>.

7) Without Feature T441, "ABS and MOD functions", conforming language shall not contain a <modulus expression>.

8) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <natural logarithm>.

9) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain an <exponential function>.

10) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <power function>.

11) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <square root>.

12) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <floor function>.

13) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <ceiling function>.

14) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <width bucket function>.

6.28 <string value expression>

Function

Specify a character string value or a binary string value.

Format

```
<string value expression> ::=
    <character value expression>
  | <blob value expression>

<character value expression> ::=
    <concatenation>
  | <character factor>

<concatenation> ::= <character value expression> <concatenation operator> <character factor>

<character factor> ::= <character primary> [ <collate clause> ]

<character primary> ::=
    <value expression primary>
  | <string value function>

<blob value expression> ::=
    <blob concatenation>
  | <blob factor>

<blob factor> ::= <blob primary>

<blob primary> ::=
    <value expression primary>
  | <string value function>

<blob concatenation> ::= <blob value expression> <concatenation operator> <blob factor>
```

Syntax Rules

1) The declared type of a <character primary> shall be character string.

2) Character strings of different character repertoires shall not be mixed in a <character value expression>.

3) The character set of a <character value expression> is that character set of its character string operands that has the character encoding form with the highest precedence.

4) Case:

   a) If <concatenation> is specified, then:

      Let *D1* be the declared type of the <character value expression> and let *D2* be the declared type of the <character factor>. Let *M* be the length in characters of *D1* plus the length in characters of *D2*. Let *VL* be the implementation-defined maximum length of variable-length character strings, let *LOL* be the

implementation-defined maximum length of large object character strings, and let *FL* be the implementation-defined maximum length of fixed-length character strings.

Case:

i) If the declared type of the <character value expression> or <character factor> is a character large object type, then the declared type of the <concatenation> is a character large object type with maximum length equal to the lesser of *M* and *LOL*.

ii) If the declared type of the <character value expression> or <character factor> is variable-length character string, then the declared type of the <concatenation> is variable-length character string with maximum length equal to the lesser of *M* and *VL*.

iii) If the declared type of the <character value expression> and <character factor> is fixed-length character string, then *M* shall not be greater than *FL* and the declared type of the <concatenation> is fixed-length character string with length *M*.

b) Otherwise, the declared type of the <character value expression> is the declared type of the <character factor>.

5) Case:

a) If <character factor> is specified, then

Case:

i) If <collate clause> is specified, then the declared type collation of the <character value expression> is the collation identified by <collate clause>, and its collation derivation is *explicit*.

ii) Otherwise, the declared type of the <character factor> is the declared type of the <character primary>.

b) If <concatenation> is specified, then its declared type is determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of its operands.

6) The declared type of <blob primary> shall be binary string.

7) If <blob concatenation> is specified, then let *M* be the length in octets of the <blob value expression> plus the length in octets of the <blob factor> and let *VL* be the implementation-defined maximum length of a binary string. The declared type of <blob concatenation> is binary string with maximum length equal to the lesser of *M* and *VL*.

Access Rules

None.

General Rules

1) If the value of any <character primary> or <blob primary> simply contained in a <string value expression> is the null value, then the result of the <string value expression> is the null value.

2) If <concatenation> is specified, then:

a) If the character repertoire of <character factor> is UCS, then, in the remainder of this General Rule, the term "length" shall be taken to mean "length in characters".

b) Let *S1* and *S2* be the result of the <character value expression> and <character factor>, respectively.

Case:

i) If either *S1* or *S2* is the null value, then the result of the <concatenation> is the null value.

ii) Otherwise:

1) Let *S* be the string consisting of *S1* followed by *S2* and let *M* be the length of *S*.

2) If the character repertoire of <character factor> is UCS, then *S* is replaced by:

Case:

A) If the <search condition> `S1 IS NORMALIZED AND S2 IS NORMALIZED` evaluates to *True*, then

```
NORMALIZE (S)
```

B) Otherwise, an implementation-defined string.

3) Case:

A) If the most specific type of either *S1* or *S2* is a character large object type, then let *LOL* be the implementation-defined maximum length of large object character strings.

Case:

I) If *M* is less than or equal to *LOL*, then the result of the <concatenation> is *S* with length *M*.

II) If *M* is greater than *LOL* and the right-most *M–LOL* characters of *S* are all the <space> character, then the result of the <concatenation> is the first *LOL* characters of *S* with length *LOL*.

III) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

B) If the most specific type of either *S1* or *S2* is variable-length character string, then let *VL* be the implementation-defined maximum length of variable-length character strings.

Case:

I) If *M* is less than or equal to *VL*, then the result of the <concatenation> is *S* with length *M*.

II) If *M* is greater than *VL* and the right-most *M–VL* characters of *S* are all the <space> character, then the result of the <concatenation> is the first *VL* characters of *S* with length *VL*.

III) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

C) If the most specific types of both *S1* and *S2* are fixed-length character string, then the result of the <concatenation> is *S*.

3) If <blob concatenation> is specified, then let *S1* and *S2* be the result of the <blob value expression> and <blob factor>, respectively.

Case:

a) If either *S1* or *S2* is the null value, then the result of the <blob concatenation> is the null value.

b) Otherwise, let *S* be the string consisting of *S1* followed by *S2* and let *M* be the length in octets of *S*.

Case:

i) If *M* is less or equal to *VL*, then the result of the <blob concatenation> is *S* with length *M*.

ii) If *M* is greater than *VL* and the right-most *M–VL* octets of *S* are all X'00', then the result of the <blob concatenation> is the first *VL* octets of *S* with length *VL*.

iii) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

4) If the result of the <character value expression> is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string*.

Conformance Rules

None.

6.29 <string value function>

Function

Specify a function yielding a value of type character string or binary string.

Format

```
<string value function> ::=
    <character value function>
  | <blob value function>

<character value function> ::=
    <character substring function>
  | <regular expression substring function>
  | <fold>
  | <transcoding>
  | <character transliteration>
  | <trim function>
  | <character overlay function>
  | <normalize function>
  | <specific type method>

<character substring function> ::=
    SUBSTRING <left paren> <character value expression> FROM <start position>
    [ FOR <string length> ] [ USING <char length units> ] <right paren>

<regular expression substring function> ::=
    SUBSTRING <left paren> <character value expression> SIMILAR <character value expression>
    ESCAPE <escape character> <right paren>

<fold> ::= { UPPER | LOWER } <left paren> <character value expression> <right paren>

<transcoding> ::=
    CONVERT <left paren> <character value expression>
    USING <transcoding name> <right paren>

<character transliteration> ::=
    TRANSLATE <left paren> <character value expression>
    USING <transliteration name> <right paren>

<trim function> ::= TRIM <left paren> <trim operands> <right paren>

<trim operands> ::= [ [ <trim specification> ] [ <trim character> ] FROM ] <trim source>

<trim source> ::= <character value expression>

<trim specification> ::=
    LEADING
  | TRAILING
  | BOTH

<trim character> ::= <character value expression>

<character overlay function> ::=
```

```
    OVERLAY <left paren> <character value expression> PLACING <character value expression>
    FROM <start position> [ FOR <string length> ]
    [ USING <char length units> ] <right paren>

<normalize function> ::= NORMALIZE <left paren> <character value expression> <right paren>

<specific type method> ::=
    <user-defined type value expression> <period> SPECIFICTYPE
    [ <left paren> <right paren> ]

<blob value function> ::=
    <blob substring function>
  | <blob trim function>
  | <blob overlay function>

<blob substring function> ::=
    SUBSTRING <left paren> <blob value expression> FROM <start position>
    [ FOR <string length> ] <right paren>

<blob trim function> ::= TRIM <left paren> <blob trim operands> <right paren>

<blob trim operands> ::=
    [ [ <trim specification> ] [ <trim octet> ] FROM ] <blob trim source>

<blob trim source> ::= <blob value expression>

<trim octet> ::= <blob value expression>

<blob overlay function> ::=
    OVERLAY <left paren> <blob value expression> PLACING <blob value expression>
    FROM <start position> [ FOR <string length> ] <right paren>

<start position> ::= <numeric value expression>

<string length> ::= <numeric value expression>
```

Syntax Rules

1) The declared type of <string value function> is the declared type of the immediately contained <character value function> or <blob value function>.

2) The declared type of <character value function> is the declared type of the immediately contained <character substring function>, <regular expression substring function>, <fold>, <transcoding>, <character transliteration>, <trim function>, <character overlay function>, <normalize function>, or <specific type method>.

3) The declared type of a <start position> and <string length> shall be exact numeric with scale 0 (zero).

4) If <character substring function> *CSF* is specified, then let *DTCVE* be the declared type of the <character value expression> immediately contained in *CSF*. The maximum length, character set, and collation of the declared type *DTCSF* of *CSF* are determined as follows:

   a) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then *DTCSF* is a variable-length character string type with maximum length equal to the fixed length or maximum length of *DTCVE*.

ii) Otherwise, the *DTCSF* is a large object character string type with maximum length equal to the maximum length of *DTCVE*.

b) The character set and collation of the <character substring function> are those of *DTCVE*.

5) If the character repertoire of <character value expression> is not UCS, then <char length units> shall not be specified.

6) If USING <char length units> is not specified, then USING CHARACTERS is implicit.

7) If <regular expression substring function> is specified, then:

a) The declared types of the <escape character> and the <character value expression>s of the <regular expression substring function> shall be character string with the same character repertoire.

b) Case:

i) If the declared type of the first <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <regular expression substring function> is variable-length character string with maximum length equal to the maximum variable length of the first <character value expression>.

ii) Otherwise, the declared type of the <regular expression substring function> is a character large object type with maximum length equal to the maximum variable length of the first <character value expression>.

c) The declared type of the <regular expression substring function> is that of the first <character value expression>.

d) The value of the <escape character> shall have length 1 (one).

8) If <fold> is specified, then the declared type of the result of <fold> is that of the <character value expression>.

9) If <transcoding> is specified, then:

a) <transcoding> shall be simply contained in a <value expression> that is immediately contained in a <derived column> that is immediately contained in a <select sublist> or shall immediately contain either a <simple value specification> that is a <host parameter name> or a <value specification> that is a <host parameter specification>.

b) A <transcoding name> shall identify a transcoding.

c) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the result is variable-length character string with implementation-defined maximum length.

ii) Otherwise, the declared type of the result is a character large object type with implementation-defined maximum length.

d) The character set of the result is an implementation-defined character set *CS* whose character repertoire is the same as the character repertoire of the <character value expression> and whose character encoding form is that determined by the transcoding identified by the <transcoding name>. The declared type collation of the result is the character set collation of *CS*.

10) If <character transliteration> is specified, then:

a) A <transliteration name> shall identify a character transliteration.

b) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <character transliteration> is variable-length character string with implementation-defined maximum length.

ii) Otherwise, the declared type of the <character transliteration> is a character large object type with implementation-defined maximum length.

c) The declared type of the <character transliteration> has the character set *CS* that is the target character set of the transliteration. The declared type collation of the result is the character set collation of *CS*.

11) If <trim function> is specified, then

a) Case:

i) If FROM is specified, then:

1) Either <trim specification> or <trim character> or both shall be specified.

2) If <trim specification> is not specified, then BOTH is implicit.

3) If <trim character> is not specified, then `' '` is implicit.

ii) Otherwise, let *SRC* be <trim source>. `TRIM ( SRC )` is equivalent to `TRIM ( BOTH ' ' FROM SRC )`.

b) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <trim function> is variable-length character string with maximum length equal to the fixed length or maximum variable length of the <trim source>.

ii) Otherwise, the declared type of the <trim function> is a character large object type with maximum length equal to the maximum variable length of the <trim source>.

c) If a <trim character> is specified, then <trim character> and <trim source> shall be comparable.

d) The declared type of the <trim function> is that of the <trim source>.

12) If <character overlay function> is specified, then:

a) Let *CV* be the first <character value expression>, let *SP* be the <start position>, and let *RS* be the second <character value expression>.

b) If <string length> is specified, then let *SL* be <string length>; otherwise, let *SL* be CHAR_LENGTH(*RS*).

c) The <character overlay function> is equivalent to:

```
    SUBSTRING ( CV FROM 1 FOR SP - 1 )
||  RS
||  SUBSTRING ( CV FROM SP + SL  )
```

13) If <normalize function> is specified, then the declared type of the result is the declared type of <string value expression>.

14) If <specific type method> is specified, then the declared type of the <specific type method> is variable-length character string with maximum length implementation-defined. The character set of the character string is SQL_IDENTIFIER.

15) The declared type of <blob value function> is the declared type of the immediately contained <blob substring function>, <blob trim function>, or <blob overlay function>.

16) If <blob substring function> is specified, then the declared type of the <blob substring function> is binary string with maximum length equal to the maximum length of the <blob value expression>.

17) If <blob trim function> is specified, then:

a) Case:

i) If FROM is specified, then:

1) Either <trim specification> or <trim octet> or both shall be specified.

2) If <trim specification> is not specified, then BOTH is implicit.

3) If <trim octet> is not specified, then X'00' is implicit.

ii) Otherwise, let *SRC* be <trim source>. `TRIM ( SRC )` is equivalent to `TRIM ( BOTH X'00' FROM SRC )`.

b) The declared type of the <blob trim function> is binary string with maximum length equal to the maximum length of the <blob trim source>.

18) If <blob overlay function> is specified, then:

a) Let *BV* be the first <blob value expression>, let *SP* be the <start position>, and let *RS* be the second <blob value expression>.

b) If <string length> is specified, then let *SL* be <string length>; otherwise, let *SL* be OCTET_LENGTH(*RS*).

c) The <blob overlay function> is equivalent to:

```
    SUBSTRING ( BV FROM 1 FOR SP - 1 )
||  RS
||  SUBSTRING ( BV FROM SP + SL  )
```

Access Rules

1) Case:

a) If <string value function> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE for every transliteration identified by a <transliteration name> contained in the <string value expression>.

b) Otherwise, the current privileges shall include USAGE for every transliteration identified by a <transliteration name> contained in the <string value expression>.

NOTE 115 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

General Rules

1) The result of <string value function> is the result of the immediately contained <character value function> or <blob value function>.

2) The result of <character value function> is the result of the immediately contained <character substring function>, <regular expression substring function>, <fold>, <transcoding>, <character transliteration>, <trim function>, <character overlay function>, or <specific type method>.

3) If <character substring function> is specified, then:

   a) If the character encoding form of <character value expression> is UTF8, UTF16, or UTF32, then, in the remainder of this General Rule, the term "character" shall be taken to mean "unit specified by <char length units>".

   b) Let *C* be the value of the <character value expression>, let *LC* be the length in characters of *C*, and let *S* be the value of the <start position>.

   c) If <string length> is specified, then let *L* be the value of <string length> and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.

   d) If either *C*, *S*, or *L* is the null value, then the result of the <character substring function> is the null value.

   e) If *E* is less than *S*, then an exception condition is raised: *data exception — substring error.*

   f) Case:

      i) If *S* is greater than *LC* or if *E* is less than 1 (one), then the result of the <character substring function> is a zero-length string.

      ii) Otherwise,

         1) Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*–*S1*.

         2) The result of the <character substring function> is a character string containing the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

4) If <normalize function> is specified, then the result is the value of <string value expression> in the normalized form of the result, in accordance with Unicode Standard Annex #15 Unicode Normalization Forms.

5) If <regular expression substring function> is specified, then:

a) Let *C* be the result of the first <character value expression>, let *R* be the result of the second <character value expression>, and let *E* be the result of the <escape character>.

b) If one or more of *C*, *R* or *E* is the null value, then the result of the <regular expression substring function> is the null value.

c) If the length in characters of *E* is not equal to 1 (one), then an exception condition is raised: *data exception — invalid escape character.*

d) If *R* does not contain exactly two occurrences of the two-character sequence consisting of *E*, each immediately followed by <double quote>, then an exception condition is raised: *data exception — invalid use of escape character.*

e) Let *R1*, *R2*, and *R3* be the substrings of *R*, such that

```
'R' = 'R1' || 'E' || '"' || 'R2' || 'E' || '"' || 'R3'
```

is *True*.

f) If any one of *R1*, *R2*, or *R3* is not a zero-length string and does not have the format of a <regular expression>, then an exception condition is raised: *data exception — invalid regular expression.*

g) If the predicate

```
'C' SIMILAR TO 'R1' || 'R2' || 'R3' ESCAPE 'E'
```

is not *True*, then the result of the <regular expression substring function> is the null value.

h) Otherwise, the result *S* of the <regular expression substring function> is computed as follows:

i) Let *S1* be the shortest initial substring of *C* such that there is a substring *S23* of *C* such that the following <search condition> is *True*:

```
'C' = 'S1' || 'S23' AND
'S1' SIMILAR TO 'R1' ESCAPE 'E' AND
'S23' SIMILAR TO '(R2R3)' ESCAPE 'E'
```

ii) Let *S3* be the shortest final substring of *S23* such that there is a substring *S2* of *S23* such that the following <search condition> is *True*:

```
'S23' = 'S2' || 'S3' AND
'S2' SIMILAR TO 'R2' ESCAPE 'E' AND
'S3' SIMILAR TO 'R3' ESCAPE 'E'
```

iii) The result of the <regular expression substring function> is *S2*.

6) If <fold> is specified, then:

a) Let *S* be the value of the <character value expression>.

b) If *S* is the null value, then the result of the <fold> is the null value.

c) Let *FRML* be the length or maximum length in characters of the declared type of <fold>.

d) Case:

i) If UPPER is specified, then let *FR* be a copy of *S* in which every lower case character that has a corresponding upper case character or characters in the character set of *S* and every title case character that has a corresponding upper case character or characters in the character set of *S* is replaced by that upper case character or characters.

ii) If LOWER is specified, then let *FR* be a copy of *S* in which every upper case character that has a corresponding lower case character or characters in the character set of *S* and every title case character that has a corresponding lower case character or characters in the character set of *S* is replaced by that lower case character or characters.

e) If the character set of <character factor> is UTF8, UTF16, or UTF32, then *FR* is replaced by

Case:

i) If the <search condition> `S IS NORMALIZED` evaluated to *True*, then
`NORMALIZE (FR)`

ii) Otherwise, *FR*.

f) Let *FRL* be the length in characters of *FR*.

g) Case:

i) If *FRL* is less than or equal to *FRML*, then the result of the <fold> is *FR*. If the declared type of *FR* is fixed-length character string, then the result is padded on the right with (*FRML* – *FRL*) <space>s.

ii) If *FRL* is greater than *FRML*, then the result of the <fold> is the first *FRML* characters of *FR* with length *FRML*. If any of the right-most (*FRL* – *FRML*) characters of *FR* are not <space> characters, then a completion condition is raised: *warning — string data, right truncation.*

7) If a <character transliteration> is specified, then

Case:

a) If the value of <character value expression> is the null value, then the result of the <character transliteration> is the null value.

b) If <transliteration name> identifies a transliteration descriptor whose indication of how the transliteration is performed specifies an SQL-invoked routine *TR*, then the result of the <character transliteration> is the result of the invocation of *TR* with a single SQL argument that is the <character value expression> contained in the <character transliteration>.

c) Otherwise, the value of the <character transliteration> is the value returned by the transliteration identified by the <existing transliteration name> specified in the transliteration descriptor of the transliteration identified by <transliteration name>.

8) If a <transcoding> is specified, then

Case:

a) If the value of <character value expression> is the null value, then the result of the <transcoding> is the null value.

b) Otherwise, the value of the <transcoding> is the value of the <character value expression> after the application of the transcoding specified by <transcoding name>.

9) If <trim function> is specified, then:

a) Let *S* be the value of the <trim source>.

b) If <trim character> is specified, then let *SC* be the value of <trim character>; otherwise, let *SC* be <space>.

c) If either *S* or *SC* is the null value, then the result of the <trim function> is the null value.

d) If the length in characters of *SC* is not 1 (one), then an exception condition is raised: *data exception — trim error*.

e) Case:

i) If BOTH is specified or if no <trim specification> is specified, then the result of the <trim function> is the value of *S* with any leading or trailing characters equal to *SC* removed.

ii) If TRAILING is specified, then the result of the <trim function> is the value of *S* with any trailing characters equal to *SC* removed.

iii) If LEADING is specified, then the result of the <trim function> is the value of *S* with any leading characters equal to *SC* removed.

10) If <specific type method> is specified, then:

a) Let *V* be the value of the <user-defined type value expression>.

b) Case:

i) If *V* is the null value, then *RV* is the null value.

ii) Otherwise:

1) Let *UDT* be the most specific type of *V*.

2) Let *UDTN* be the <user-defined type name> of *UDT*.

3) Let *CN* be the <catalog name> contained in *UDTN*, let *SN* be the <unqualified schema name> contained in *UDTN*, and let *UN* be the <qualified identifier> contained in *UDTN*. Let *CND*, *SND*, and *UND* be *CN*, *SN*, and *UN*, respectively, with every occurrence of <double quote> replaced by <doublequote symbol>. Let *RV* be:

"*CND*" . "*SND*" . "*UND*"

c) The result of <specific type method> is *RV*.

11) The result of <blob value function> is the result of the simply contained <blob substring function>, <blob trim function>, or <blob overlay function>.

12) If <blob substring function> is specified, then

a) Let *B* be the value of the <blob value expression>, let *LB* be the length in octets of *B*, and let *S* be the value of the <start position>.

b) If <string length> is specified, then let *L* be the value of <string length> and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LB*+1 and *S*.

c) If either *B*, *S*, or *L* is the null value, then the result of the <blob substring function> is the null value.

d) If *E* is less than *S*, then an exception condition is raised: *data exception — substring error*.

e) Case:

i) If *S* is greater than *LB* or if *E* is less than 1 (one), then the result of the <blob substring function> is a zero-length string.

ii) Otherwise:

1) Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LB*+1. Let *L1* be *E1*–*S1*.

2) The result of the <blob substring function> is a binary large object string containing *L1* octets of *B* starting at octet number *S1* in the same order that the octets appear in *B*.

13) If <blob trim function> is specified, then

a) Let *S* be the value of the <trim source>.

b) Let *SO* be the value of <trim octet>.

c) If either *S* or *SO* the null value, then the result of the <blob trim function> is the null value.

d) If the length in octets of *SO* is not 1 (one), then an exception condition is raised: *data exception — trim error*.

e) Case:

i) If BOTH is specified or if no <trim specification> is specified, then the result of the <blob trim function> is the value of *S* with any leading or trailing octets equal to *SO* removed.

ii) If TRAILING is specified, then the result of the <blob trim function> is the value of *S* with any trailing octets equal to *SO* removed.

iii) If LEADING is specified, then the result of the <blob trim function> is the value of *S* with any leading octets equal to *SO* removed.

14) If the result of <string value expression> is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string*.

Conformance Rules

1) Without Feature T581, "Regular expression substring function", conforming SQL language shall not contain a <regular expression substring function>.

2) Without Feature T312, "OVERLAY function", conforming SQL language shall not contain a <character overlay function>.

3) Without Feature T312, "OVERLAY function", conforming SQL language shall not contain a <blob overlay function>.

4) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain a <blob value function>.

5) Without Feature F695, "Translation support", conforming SQL language shall not contain a <character transliteration>.

6) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transcoding>.

7) Without Feature T061, "UCS support", conforming SQL language shall not contain a <normalize function>.

8) Without Feature S261, "Specific type method", conforming SQL language shall not contain a <specific type method>.

6.30 <datetime value expression>

Function

Specify a datetime value.

Format

```
<datetime value expression> ::=
    <datetime term>
  | <interval value expression> <plus sign> <datetime term>
  | <datetime value expression> <plus sign> <interval term>
  | <datetime value expression> <minus sign> <interval term>

<datetime term> ::= <datetime factor>

<datetime factor> ::= <datetime primary> [ <time zone> ]

<datetime primary> ::=
    <value expression primary>
  | <datetime value function>

<time zone> ::= AT <time zone specifier>

<time zone specifier> ::=
    LOCAL
  | TIME ZONE <interval primary>
```

Syntax Rules

1) The declared type of a <datetime primary> shall be datetime.

2) If the <datetime value expression> immediately contains neither <plus sign> nor <minus sign>, then the precision of the result of the <datetime value expression> is the precision of the <value expression primary> or <datetime value function> that it simply contains.

3) If the declared type of the <datetime primary> is DATE, then <time zone> shall not be specified.

4) Case:

   a) If <time zone> is specified and the declared type of <datetime primary> is TIMESTAMP WITHOUT TIME ZONE or TIME WITHOUT TIME ZONE, then the declared type of <datetime term> is TIMESTAMP WITH TIME ZONE or TIME WITH TIME ZONE, respectively, with the same fractional seconds precision as <datetime primary>.

   b) Otherwise, the declared type of <datetime term> is the same as the declared type of <datetime primary>.

5) If the <datetime value expression> immediately contains either <plus sign> or <minus sign>, then:

   a) The <interval value expression> or <interval term> shall contain only <primary datetime field>s that are contained within the <datetime value expression> or <datetime term>.

b) The result of the <datetime value expression> contains the same <primary datetime field>s that are contained in the <datetime value expression> or <datetime term>, with a fractional seconds precision that is the greater of the fractional seconds precisions, if any, of either the <datetime value expression> and <interval term>, or the <datetime term> and <interval value expression> that it simply contains.

6) The declared type of the <interval primary> immediately contained in a <time zone specifier> shall be INTERVAL HOUR TO MINUTE.

Access Rules

None.

General Rules

1) If the value of any <datetime primary>, <interval value expression>, <datetime value expression>, or <interval term> simply contained in a <datetime value expression> is the null value, then the result of the <datetime value expression> is the null value.

2) If <time zone> is specified and the <interval primary> immediately contained in <time zone specifier> is null, then the result of the <datetime value expression> is the null value.

3) The value of a <datetime primary> is the value of the immediately contained <value expression primary> or <datetime value function>.

4) In the following General Rules, arithmetic is performed so as to maintain the integrity of the datetime data type that is the result of the <datetime term> or <datetime value expression>. This may involve carry from or to the immediately next more significant <primary datetime field>. If the data type of the <datetime term> or <datetime value expression> is time with or without time zone, then arithmetic on the HOUR <primary datetime field> is undertaken modulo 24. If the <interval value expression> or <interval term> is a year-month interval, then the DAY field of the result is the same as the DAY field of the <datetime term> or <datetime value expression>.

5) The value of a <datetime term> is determined as follows. Let *DT* be the declared type, *DV* the UTC component of the value, and *TZD* the time zone component, if any, of the <datetime primary> simply contained in the <datetime term>, and let *STZD* be the current default time zone displacement of the SQL-session.

   Case:

   a) If <time zone> is not specified, then the value of <datetime term> is *DV*.

   b) Otherwise:

      i) Case:

         1) If *DT* is datetime with time zone, then the UTC component of the <datetime term> is *DV*.

         2) Otherwise, the UTC component of the <datetime term> is *DV* – *STZD*.

      ii) Case:

         1) If LOCAL is specified, then let *TZ* be *STZD*.

2) If TIME ZONE is specified, then, if the value of the <interval primary> immediately contained in <time zone specifier> is less than `INTERVAL -'12:59'` or greater than `INTERVAL +'14:00'`, then an exception condition is raised: *data exception — invalid time zone displacement value*. Otherwise, let *TZ* be the value of the <interval primary> simply contained in <time zone>.

iii) The time zone component of the value of the <datetime term> is *TZ*.

6) If a <datetime value expression> immediately contains the operator <plus sign> or <minus sign>, then the time zone component, if any, of the result is the same as the time zone component of the immediately contained <datetime term> or <datetime value expression>. The result (if the result type is without time zone) or the UTC component of the result (if the result type has time zone) is effectively evaluated as follows:

a) Case:

i) If <datetime value expression> immediately contains the operator <plus sign> and the <interval value expression> or <interval term> is not negative, or if <datetime value expression> immediately contains the operator <minus sign> and the <interval term> is negative, then successive <primary datetime field>s of the <interval value expression> or <interval term> are added to the corresponding fields of the <datetime value expression> or <datetime term>.

ii) Otherwise, successive <primary datetime field>s of the <interval value expression> or <interval term> are subtracted from the corresponding fields of the <datetime value expression> or <datetime term>.

b) If, after the preceding step, any <primary datetime field> of the result is outside the permissible range of values for the field or the result is invalid based on the natural rules for dates and times, then an exception condition is raised: *data exception — datetime field overflow*.

NOTE 116 — For the permissible range of values for <primary datetime field>s, see Table 9, "Valid values for datetime fields".

Conformance Rules

1) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain <datetime value expression> that immediately contains a <plus sign> or a <minus sign>.

2) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <time zone>.

6.31 <datetime value function>

Function

Specify a function yielding a value of type datetime.

Format

```
<datetime value function> ::=
    <current date value function>
  | <current time value function>
  | <current timestamp value function>
  | <current local time value function>
  | <current local timestamp value function>

<current date value function> ::= CURRENT_DATE

<current time value function> ::=
    CURRENT_TIME [ <left paren> <time precision> <right paren> ]

<current local time value function> ::=
    LOCALTIME [ <left paren> <time precision> <right paren> ]

<current timestamp value function> ::=
    CURRENT_TIMESTAMP [ <left paren> <timestamp precision> <right paren> ]

<current local timestamp value function> ::=
    LOCALTIMESTAMP [ <left paren> <timestamp precision> <right paren> ]
```

Syntax Rules

1) The declared type of a <current date value function> is DATE. The declared type of a <current time value function> is TIME WITH TIME ZONE. The declared type of a <current timestamp value function> is TIMESTAMP WITH TIME ZONE.

   NOTE 117 — See the Syntax Rules of Subclause 6.1, "<data type>", for rules governing <time precision> and <timestamp precision>.

2) If <time precision> *TP* is specified, then LOCALTIME(*TP*) is equivalent to:

   ```
   CAST (CURRENT_TIME(TP) AS TIME(TP) WITHOUT TIME ZONE)
   ```

   Otherwise, LOCALTIME is equivalent to:

   ```
   CAST (CURRENT_TIME AS TIME WITHOUT TIME ZONE)
   ```

3) If <timestamp precision> *TP* is specified, then LOCALTIMESTAMP(*TP*) is equivalent to:

   ```
   CAST (CURRENT_TIMESTAMP(TP) AS TIMESTAMP(TP) WITHOUT TIME ZONE)
   ```

   Otherwise, LOCALTIMESTAMP is equivalent to:

   ```
   CAST (CURRENT_TIMESTAMP AS TIMESTAMP WITHOUT TIME ZONE)
   ```

Access Rules

None.

General Rules

1) The <datetime value function>s CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP respectively return the current date, current time, and current timestamp; the time and timestamp values are returned with time zone displacement equal to the current default time zone displacement of the SQL-session.

2) If specified, <time precision> and <timestamp precision> respectively determine the precision of the time or timestamp value returned.

3) Let *S* be an <SQL procedure statement> that is not generally contained in a <triggered action>. All <datetime value function>s that are contained in <value expression>s that are generally contained, without an intervening <routine invocation> whose subject routines do not include an SQL function, either in *S* without an intervening <SQL procedure statement> or in an <SQL procedure statement> contained in the <triggered action> of a trigger activated as a consequence of executing *S*, are effectively evaluated simultaneously. The time of evaluation of a <datetime value function> during the execution of *S* and its activated triggers is implementation-dependent.

   NOTE 118 — Activation of triggers is defined in Subclause 4.38.2, "Trigger execution".

Conformance Rules

1) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <current local time value function> that contains a <time precision> that is not 0 (zero).

2) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <current local timestamp value function> that contains a <timestamp precision> that is neither 0 (zero) nor 6.

3) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <current time value function>.

4) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <current timestamp value function>.

6.32 <interval value expression>

Function

Specify an interval value.

Format

```
<interval value expression> ::=
    <interval term>
  | <interval value expression 1> <plus sign> <interval term 1>
  | <interval value expression 1> <minus sign> <interval term 1>
  | <left paren> <datetime value expression> <minus sign> <datetime term> <right paren>
    <interval qualifier>

<interval term> ::=
    <interval factor>
  | <interval term 2> <asterisk> <factor>
  | <interval term 2> <solidus> <factor>
  | <term> <asterisk> <interval factor>

<interval factor> ::= [ <sign> ] <interval primary>

<interval primary> ::=
    <value expression primary> [ <interval qualifier> ]
  | <interval value function>

<interval value expression 1> ::= <interval value expression>

<interval term 1> ::= <interval term>

<interval term 2> ::= <interval term>
```

Syntax Rules

1) The declared type of an <interval value expression> is interval. The declared type of a <value expression primary> immediately contained in an <interval primary> shall be interval.

2) Case:

   a) If the <interval value expression> simply contains an <interval qualifier> *IQ*, then the declared type of the result is INTERVAL *IQ*.

   b) If the <interval value expression> is an <interval term>, then the result of the <interval value expression> contains the same interval fields as the <interval primary>. If the <interval primary> contains a seconds field, then the result's fractional seconds precision is the same as the <interval primary>'s fractional seconds precision. The result's <interval leading field precision> is implementation-defined, but shall not be less than the <interval leading field precision> of the <interval primary>.

   c) If <interval term 1> is specified, then the result contains every interval field that is contained in the result of either <interval value expression 1> or <interval term 1>, and, if both contain a seconds field, then the fractional seconds precision of the result is the greater of the two fractional seconds precisions.

The <interval leading field precision> is implementation-defined, but shall be sufficient to represent all interval values with the interval fields and <interval leading field precision> of <interval value expression 1> as well as all interval values with the interval fields and <interval leading field precision> of <interval term 1>.

NOTE 119 — Interval fields are effectively defined by Table 4, "Fields in year-month INTERVAL values", and Table 5, "Fields in day-time INTERVAL values".

3) Case:

   a) If <interval term 1> is a year-month interval, then <interval value expression 1> shall be a year-month interval.

   b) If <interval term 1> is a day-time interval, then <interval value expression 1> shall be a day-time interval.

4) If <datetime value expression> is specified, then <datetime value expression> and <datetime term> shall be comparable.

5) An <interval primary> shall specify <interval qualifier> only if the <interval primary> specifies a <dynamic parameter specification>.

Access Rules

None.

General Rules

1) If an <interval term> specifies "<term> * <interval factor>", then let *T* and *F* be respectively the value of the <term> and the value of the <interval factor>. The result of the <interval term> is the result of *F* * *T*.

2) If the value of any <interval primary>, <datetime value expression>, <datetime term>, or <factor> that is simply contained in an <interval value expression> is the null value, then the result of the <interval value expression> is the null value.

3) If *IP* is an <interval primary>, then

   Case:

   a) If *IP* immediately contains a <value expression primary> *VEP* and an explicit <interval qualifier> *IQ*, then the value of *IP* is computed by:

```
CAST ( VEP AS INTERVAL IQ )
```

   b) If *IP* immediately contains a <value expression primary> *VEP*, then the value of *IP* is the value of *VEP*.

   c) If *IP* is an <interval value function> *IVF*, then the value of *IP* is the value of *IVF*.

4) If the <sign> of an <interval factor> is <minus sign>, then the value of the <interval factor> is the negative of the value of the <interval primary>; otherwise, the value of an <interval factor> is the value of the <interval primary>.

5) If <interval term 2> is specified, then:

a) Let *X* be the value of <interval term 2> and let *Y* be the value of <factor>.

b) Let *P* and *Q* be respectively the most significant and least significant <primary datetime field>s of <interval term 2>.

c) Let *E* be an exact numeric result of the operation

```
CAST ( CAST ( X AS INTERVAL Q ) AS E1 )
```

where *E1* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

d) Let *OP* be the operator * or / specified in the <interval value expression>.

e) Let *I*, the result of the <interval value expression> expressed in terms of the <primary datetime field> *Q*, be the result of

```
CAST ( (E OP Y) AS INTERVAL Q )
```

f) The result of the <interval value expression> is

```
CAST ( I AS INTERVAL W )
```

where *W* is an <interval qualifier> identifying the <primary datetime field>s *P* TO *Q*, but with <interval leading field precision> such that significant digits are not lost.

6) If <interval term 1> is specified, then let *P* and *Q* be respectively the most significant and least significant <primary datetime field>s in <interval term 1> and <interval value expression 1>, let *X* be the value of <interval value expression 1>, and let *Y* be the value of <interval term 1>.

a) Let *A* be an exact numeric result of the operation

```
CAST ( CAST ( X AS INTERVAL Q ) AS E1 )
```

where *E1* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

b) Let *B* be an exact numeric result of the operation

```
CAST ( CAST ( Y AS INTERVAL Q ) AS E2 )
```

where *E2* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

c) Let *OP* be the operator + or – specified in the <interval value expression>.

d) Let *I*, the result of the <interval value expression> expressed in terms of the <primary datetime field> *Q*, be the result of:

```
CAST ( (A OP B) AS INTERVAL Q )
```

e) The result of the <interval value expression> is

```
CAST ( I AS INTERVAL W )
```

where *W* is an <interval qualifier> identifying the <primary datetime field>s *P* TO *Q*, but with <interval leading field precision> such that significant digits are not lost.

7) If <datetime value expression> is specified, then let *Y* be the least significant <primary datetime field> specified by <interval qualifier>. Let *DTE* be the <datetime value expression>, let *DT* be the <datetime term>, and let *MSP* be the implementation-defined maximum seconds precision. Evaluation of <interval value expression> proceeds as follows:

   a) Case:

      i) If the declared type of <datetime value expression> is TIME WITH TIME ZONE, then let *A* be the value of:

         ```
         CAST ( DTE AT LOCAL AS TIME(MSP) WITHOUT TIME ZONE )
         ```

      ii) If the declared type of <datetime value expression> is TIMESTAMP WITH TIME ZONE, then let *A* be the value of:

         ```
         CAST ( DTE AT LOCAL AS TIMESTAMP(MSP) WITHOUT TIME ZONE )
         ```

      iii) Otherwise, let *A* be the value of *DTE*.

   b) Case:

      i) If the declared type of <datetime term> is TIME WITH TIME ZONE, then let *B* be the value of:

         ```
         CAST ( DT AT LOCAL AS TIME(MSP) WITHOUT TIME ZONE )
         ```

      ii) If the declared type of <datetime term> is TIMESTAMP WITH TIME ZONE, then let *B* be the value of:

         ```
         CAST ( DT AT LOCAL AS TIMESTAMP(MSP) WITHOUT TIME ZONE )
         ```

      iii) Otherwise, let *B* be the value of *DTE*.

   c) *A* and *B* are converted to integer scalars *A2* and *B2* respectively in units *Y* as displacements from some implementation-dependent start datetime.

   d) The result is determined by effectively computing *A2–B2* and then converting the difference to an interval using an <interval qualifier> whose <end field> is *Y* and whose <start field> is sufficiently significant to avoid loss of significant digits. The difference of two values of type TIME (with or without time zone) is constrained to be between –24:00:00 and +24:00:00 (excluding each end point); it is implementation-defined which of two non-zero values in this range is the result, although the computation shall be deterministic. That interval is then converted to an interval using the specified <interval qualifier>, rounding or truncating if necessary. The choice of whether to round or truncate is implementation-defined. If the required number of significant digits exceeds the implementation-defined maximum number of significant digits, then an exception condition is raised: *data exception — interval field overflow.*

Conformance Rules

1) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval value expression>.

6.33 <interval value function>

Function

Specify a function yielding a value of type interval.

Format

```
<interval value function> ::= <interval absolute value function>

<interval absolute value function> ::=
    ABS <left paren> <interval value expression> <right paren>
```

Syntax Rules

1) If <interval absolute value function> is specified, then the declared type of the result is the declared type of the <interval value expression>.

Access Rules

None.

General Rules

1) If <interval absolute value function> is specified, then let *N* be the value of the <interval value expression>.

   Case:

   a) If *N* is the null value, then the result is the null value.

   b) If $N \geq 0$ (zero), then the result is *N*.

   c) Otherwise, the result is $-1 * N$.

Conformance Rules

1) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL shall not contain an <interval value function>.

6.34 <boolean value expression>

Function

Specify a boolean value.

Format

```
<boolean value expression> ::=
    <boolean term>
  | <boolean value expression> OR <boolean term>

<boolean term> ::=
    <boolean factor>
  | <boolean term> AND <boolean factor>

<boolean factor> ::= [ NOT ] <boolean test>

<boolean test> ::= <boolean primary> [ IS [ NOT ] <truth value> ]

<truth value> ::=
    TRUE
  | FALSE
  | UNKNOWN

<boolean primary> ::=
    <predicate>
  | <boolean predicand>

<boolean predicand> ::=
    <parenthesized boolean value expression>
  | <nonparenthesized value expression primary>

<parenthesized boolean value expression> ::=
    <left paren> <boolean value expression> <right paren>
```

Syntax Rules

1) The declared type of a <nonparenthesized value expression primary> shall be boolean.

2) If NOT is specified in a <boolean test>, then let *BP* be the contained <boolean primary> and let *TV* be the contained <truth value>. The <boolean test> is equivalent to:

   ```
   ( NOT ( BP IS TV ) )
   ```

3) Let *X* denote either a column *C* or the <key word> VALUE. Given a <boolean value expression> *BVE* and *X*, the notion "*BVE* is a *known-not-null condition* for *X*" is defined recursively as follows:

   a) If *BVE* is a <predicate>, then

      Case:

i) If *BVE* is a <predicate> of the form "*RVE* IS NOT NULL", where *RVE* is a <row value predicand> that is a <row value constructor predicand> that simply contains a <common value expression>, <boolean predicand>, or <row value constructor element> that is a <column reference> that references *C*, then *BVE* is a known-not-null condition for *C*.

ii) If *BVE* is the <predicate> "VALUE IS NOT NULL", then *BVE* is a known-not-null condition for VALUE.

iii) Otherwise, *BVE* is not a known-not-null condition for *X*.

b) If *BVE* is a <parenthesized boolean value expression> and the simply contained <boolean value expression> is a known-not-null condition for *X*, then *BVE* is a known-not-null condition for *X*.

c) If *BVE* is a <nonparenthesized value expression primary>, then *BVE* is not a known-not-null condition for *X*.

d) If *BVE* is a <boolean test>, then let *BP* be the <boolean primary> immediately contained in *BVE*. If *BP* is a known-not-null condition for *X*, and <truth value> is not specified, then *BVE* is a known-not-null condition for *X*. Otherwise, *BVE* is not a known-not-null condition for *X*.

e) If *BVE* is of the form "NOT *BT*", where *BT* is a <boolean test>, then

Case:

i) If *BT* is "*CR* IS NULL", where *CR* is a column reference that references column *C*, then *BVE* is a known-not-null condition for *C*.

ii) If *BT* is "VALUE IS NULL", then *BVE* is a known-not-null condition for VALUE.

iii) Otherwise, *BVE* is not a known-not-null condition for *X*.

NOTE 120 — For simplicity, this rule does not attempt to analyze conditions such as "NOT NOT *A* IS NULL", or "NOT (*A* IS NULL OR NOT (*B* = 2))"

f) If *BVE* is of the form "*BVE1* AND *BVE2*", then

Case:

i) If either *BVE1* or *BVE2* is a known-not-null condition for *X*, then *BVE* is aknown-not-null condition for *X*.

ii) Otherwise, *BVE* is not a known-not-null condition for *X*.

g) If *BVE* is of the form "*BVE1* OR *BVE2*", then *BVE* is not a known-not-null condition for *X*.

NOTE 121 — For simplicity, this rule does not detect cases such as "*A* IS NOT NULL OR *A* IS NOT NULL", which might be classified as a known-not-null condition.

4) The notion of "retrospectively deterministic" is defined recursively as follows:

a) A <parenthesized boolean value expression> is retrospectively deterministic if the simply contained <boolean value expression> is retrospectively deterministic.

b) A <nonparenthesized value expression primary> is retrospectively deterministic if it is not possibly non-deterministic.

c) A <predicate> *P* is *retrospectively deterministic* if one of the following is true:

i) *P* is not possibly non-deterministic.

ii) *P* is a <comparison predicate> of the form "*X* < *Y*", "*X* <= *Y*", "*Y* > *X*", "*Y* >= *X*", "*X* < *Y* + *Z*", "*X* <= *Y* + *Z*", "*Y* + *Z* > *X*", "*Y* + *Z* >= *X*", "*X* < *Y* – *Z*", "*X* <= *Y* – *Z*", "*Y* – *Z* > *X*", or "*Y* – *Z* >= *X*", where *Y* is CURRENT_DATE, CURRENT_TIMESTAMP or LOCALTIMESTAMP, *X* and *Z* are not possibly non-deterministic <value expression>s, and the declared types of the left and right comparands are either both datetime with time zone or both datetime without time zone.

iii) *P* is a <quantified comparison predicate> of the form "*Y* > <quantifier> <table subquery>", "*Y* + *Z* > <quantifier> <table subquery>", "*Y* – *Z* > <quantifier> <table subquery>", "*Y* >= <quantifier> <table subquery>", "*Y* + *Z* >= <quantifier> <table subquery>", or "*Y* - *Z* >= <quantifier> <table subquery>", where *Y* is CURRENT_DATE, CURRENT_TIMESTAMP or LOCALTIMESTAMP, *Z* is a <value expression> that is not possibly non-deterministic, the <query expression> simply contained in the <table subquery> is not possibly non-deterministic, and the declared types of the left and right comparands are either both datetime with time zone or both datetime without time zone.

iv) *P* is a <between predicate> that is transformed into a retrospectively deterministic <boolean value expression>.

d) A <boolean primary> is retrospectively deterministic if the simply contained <predicate>, <parenthesized boolean value expression> or <nonparenthesized value expression primary> is retrospectively deterministic.

e) Let *BF* be a <boolean factor>. Let *BP* be the <boolean primary> simply contained in *BF*.

i) *BF* is called *negative* if *BF* is of any of the following forms:

```
NOT BP
BP IS FALSE
BP IS NOT TRUE
NOT BP IS NOT FALSE
NOT BP IS TRUE
```

ii) *BF* is retrospectively deterministic if one of the following is true:

1) *BF* is negative and *BF* does not generally contain a possibly nondeterministic <value expression>.

2) *BF* is not negative and *BP* is retrospectively deterministic.

f) A <boolean value expression> is retrospectively deterministic if every simply contained <boolean factor> is retrospectively deterministic.

Access Rules

None.

General Rules

1) The result is derived by the application of the specified boolean operators ("AND", "OR", "NOT", and "IS") to the results derived from each <boolean primary>. If boolean operators are not specified, then the result of the <boolean value expression> is the result of the specified <boolean primary>.

2) NOT (*True*) is *False*, NOT (*False*) is *True*, and NOT (*Unknown*) is *Unknown*.

3) Table 11, "Truth table for the AND boolean operator", Table 12, "Truth table for the OR boolean operator", and Table 13, "Truth table for the IS boolean operator" specify the semantics of AND, OR, and IS, respectively.

Table 11 — Truth table for the AND boolean operator

AND	*True*	*False*	*Unknown*
True	*True*	*False*	*Unknown*
False	*False*	*False*	*False*
Unknown	*Unknown*	*False*	*Unknown*

Table 12 — Truth table for the OR boolean operator

OR	*True*	*False*	*Unknown*
True	*True*	*True*	*True*
False	*True*	*False*	*Unknown*
Unknown	*True*	*Unknown*	*Unknown*

Table 13 — Truth table for the IS boolean operator

IS	TRUE	FALSE	UNKNOWN
True	*True*	*False*	*False*
False	*False*	*True*	*False*
Unknown	*False*	*False*	*True*

Conformance Rules

1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean primary> that simply contains a <nonparenthesized value expression primary>.

2) Without Feature F571, "Truth value tests", conforming SQL language shall not contain a <boolean test> that simply contains a <truth value>.

6.35 <array value expression>

Function

Specify an array value.

Format

```
<array value expression> ::=
    <array concatenation>
  | <array primary>

<array concatenation> ::= <array value expression 1> <concatenation operator> <array primary>

<array value expression 1> ::= <array value expression>

<array primary> ::= <value expression primary>
```

Syntax Rules

1) The declared type of the <array value expression> is the declared type of the immediately contained <array concatenation> or <array primary>.

2) The declared type of <array primary> is the declared type of the immediately contained <value expression primary>, which shall be an array type.

3) If <array concatenation> is specified, then:

   a) Let *DT* be the data type determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of <array value expression 1> and <array primary>.

   b) Let *IMDC* be the implementation-defined maximum cardinality of an array type.

   c) The declared type of the result of <array concatenation> is an array type whose element type is the element type of *DT* and whose maximum cardinality is the lesser of *IMDC* and the sum of the maximum cardinality of <array value expression 1> and the maximum cardinality of <array primary>.

Access Rules

None.

General Rules

1) The value of the result of <array value expression> is the value of the immediately contained <array concatenation> or <array primary>.

2) If <array concatenation> is specified, then let *AV1* be the value of <array value expression 1> and let *AV2* be the value of <array primary>.

Case:

a) If either *AV1* or *AV2* is the null value, then the result of the <array concatenation> is the null value.

b) If the sum of the cardinality of *AV1* and the cardinality of *AV2* is greater than *IMDC*, then an exception condition is raised: *data exception — array data, right truncation.*

c) Otherwise, the result is the array comprising every element of *AV1* followed by every element of *AV2*.

Conformance Rules

1) Without Feature S091, "Basic array support", conforming SQL language shall not contain an <array value expression>.

6.36 <array value constructor>

Function

Specify construction of an array.

Format

```
<array value constructor> ::=
    <array value constructor by enumeration>
  | <array value constructor by query>

<array value constructor by enumeration> ::=
    ARRAY <left bracket or trigraph> <array element list> <right bracket or trigraph>

<array element list> ::=
    <array element> [ { <comma> <array element> }... ]

<array element> ::= <value expression>

<array value constructor by query> ::=
    ARRAY <left paren> <query expression> [ <order by clause> ] <right paren>
```

Syntax Rules

1) The declared type of <array value constructor> is the declared type of the immediately contained <array value constructor by enumeration> or <array value constructor by query>.

2) The declared type of the <array value constructor by enumeration> is an array type with element type *DT*, where *DT* is the declared type determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of the <array element>s immediately contained in the <array element list> of this <array value constructor by enumeration>. The maximum cardinality is the number of <array element>s in the <array element list>, which shall not be greater than the implementation-defined maximum cardinality for array types whose element type is *DT*.

3) If <array value constructor by query> is specified, then

   a) The <query expression> shall be of degree 1 (one). Let *ET* be the declared type of the column in the result of <query expression>.

   b) The declared type of the <array value constructor by query> is array with element type *ET* and maximum cardinality equal to the implementation-defined maximum cardinality *IMDC* for such array types.

Access Rules

None.

General Rules

1) The value of <array value constructor> is the value of the immediately contained <array value constructor by enumeration> or <array value constructor by query>.

2) The result of <array value constructor by enumeration> is an array whose *i*-th element is the value of the *i*-th <array element> immediately contained in the <array element list>, cast as the data type of *DT*.

3) The result of <array value constructor by query> is determined as follows:

   a) The <query expression> is evaluated, producing a table *T*. Let *N* be the number of rows in *T*.

   b) If *N* is greater than *IMDC*, then an exception condition is raised: *data exception — array data, right truncation*.

   c) *T* is ordered according to the <sort specification list>. If there is no <sort specification list>, then the ordering is implementation-dependent.

   d) The result of <array value constructor by query> is an array of *N* elements such that for all *i*, 1 (one) $\leq i \leq N$, the value of the *i*-th element is the value of the only column in the *i*-th row of *T*, as ordered by GR 3)c)the .

Conformance Rules

1) Without Feature S091, "Basic array support", conforming SQL language shall not contain an <array value constructor by enumeration>.

2) Without Feature S095, "Array constructors by query", conforming SQL language shall not contain an <array value constructor by query>.

6.37 <multiset value expression>

Function

Specify a multiset value.

Format

```
<multiset value expression> ::=
    <multiset term>
  | <multiset value expression> MULTISET UNION [ ALL | DISTINCT ] <multiset term>
  | <multiset value expression> MULTISET EXCEPT [ ALL | DISTINCT ] <multiset term>

<multiset term> ::=
    <multiset primary>
  | <multiset term> MULTISET INTERSECT [ ALL | DISTINCT ] <multiset primary>

<multiset primary> ::=
    <multiset value function>
  | <value expression primary>
```

Syntax Rules

1) The declared type of a <multiset primary> is the declared type of the immediately contained <multiset value function> or <value expression primary>, which shall be a multiset type.

2) If *MI* is a <multiset term> that immediately contains MULTISET INTERSECT, then let *OP1* be the first operand (the <multiset term>) and let *OP2* be the second operand (the <multiset primary>).

   a) *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

   b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. Let *ET* be the data type determined by Subclause 9.3, "Data types of results of aggregations", using the types *ET1* and *ET2*. The result type of the MULTISET INTERSECT operation is multiset with element type *ET*.

   c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

   d) *MI* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       INTERSECT SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

3) If *MU* is a <multiset value expression> that immediately contains MULTISET UNION, then let *OP1* be the first operand (the <multiset value expression>) and let *OP2* be the second operand (the <multiset term>).

a) If DISTINCT is specified, then *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. Let *ET* be the data type determined by Subclause 9.3, "Data types of results of aggregations", using the types *ET1* and *ET2*. The result type of the MULTISET UNION operation is multiset with element type *ET*.

c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

d) *MU* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       UNION SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

4) If *ME* is a <multiset value expression> that immediately contains MULTISET EXCEPT, then let *OP1* be the first operand (the <multiset term>) and let *OP2* be the second operand (the <multiset primary>).

a) *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. Let *ET* be the data type determined by Subclause 9.3, "Data types of results of aggregations", using the types *ET1* and *ET2*. The result type of the MULTISET EXCEPT operation is multiset with element type *ET*.

c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

d) *ME* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       EXCEPT SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

Access Rules

None.

General Rules

1) The value of a <multiset primary> is the value of the immediately contained <multiset value function> or <value expression primary>.

2) The value of a <multiset term> that is a <multiset primary> is the value of the <multiset primary>.

3) The value of a <multiset value expression> that is a <multiset term> is the value of <multiset term>.

Conformance Rules

1) Without Feature S275, "Advanced multiset support", conforming SQL language shall not contain MULTISET UNION, MULTISET INTERSECTION, or MULTISET EXCEPT.

NOTE 122 — If MULTISET UNION DISTINCT, MULTISET INTERSECTION, or MULTISET EXCEPT is specified, then the Conformance Rules of Subclause 9.11, "Multiset element grouping operations", also apply.

6.38 <multiset value function>

Function

Specify a function yielding a value of a multiset type.

Format

```
<multiset value function> ::= <multiset set function>

<multiset set function> ::=
    SET <left paren> <multiset value expression> <right paren>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression> simply contained in <multiset set function>. *MVE* is a multiset operand of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

2) The <multiset set function> is equivalent to

```
( CASE WHEN MVE IS NULL THEN NULL
       ELSE MULTISET ( SELECT DISTINCT M.E
                       FROM UNNEST (MVE) AS M(E) )
  END )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset value function>.

NOTE 123 — The Conformance Rules of Subclause 9.11, "Multiset element grouping operations", also apply.

6.39 <multiset value constructor>

Function

Specify construction of a multiset.

Format

```
<multiset value constructor> ::=
    <multiset value constructor by enumeration>
  | <multiset value constructor by query>
  | <table value constructor by query>

<multiset value constructor by enumeration> ::=
    MULTISET <left bracket or trigraph> <multiset element list> <right bracket or trigraph>

<multiset element list> ::=
    <multiset element> [ { <comma> <multiset element> }... ]

<multiset element> ::= <value expression>

<multiset value constructor by query> ::=
    MULTISET <left paren> <query expression> <right paren>

<table value constructor by query> ::=
    TABLE <left paren> <query expression> <right paren>
```

Syntax Rules

1) If <multiset value constructor> immediately contains a <table value constructor by query> *TVCBQ*, then:

   a) Let *QE* be the <query expression> simply contained in *TVCBQ*.

   b) Let *n* be the number of columns in the result of *QE*.

   c) Let C_1, ..., C_n be implementation-dependent identifiers that are all distinct from one another.

   d) *TVCBQ* is equivalent to

```
MULTISET ( SELECT ROW ( C1, ..., Cn )
FROM ( QE ) AS T ( C1, ..., Cn ) )
```

2) The declared type of <multiset value constructor> is the declared type of the immediately contained <multiset value constructor by enumeration> or <multiset value constructor by query>.

3) The declared type of the <multiset value constructor by enumeration> is a multiset type with element type *DT*, where *DT* is the declared type determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of the <multiset element>s immediately contained in the <multiset element list> of this <multiset value constructor by enumeration>.

4) If <multiset value constructor by query> is specified, then

a) The <query expression> shall be of degree 1 (one). Let *ET* be the declared type of the column in the result of <query expression>.

b) The declared type of the <multiset value constructor by query> is multiset with element type *ET*.

Access Rules

None.

General Rules

1) The value of <multiset value constructor> is the value of the immediately contained <multiset value constructor by enumeration> or <multiset value constructor by query>.

2) The result of <multiset value constructor by enumeration> is a multiset whose elements are the values of the <multiset element>s immediately contained in the <multiset element list>, cast as the data type of *DT*.

3) If <multiset value constructor by query> is specified, then:

   a) The <query expression> is evaluated, producing a table *T*. Let *N* be the number of rows in *T*.

   b) The result of <multiset value constructor by query> is a multiset of *N* elements, with one element for each row of *T*, where the value of each element is the value of the only column in the corresponding row of *T*.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset value constructor>.

2) Without Feature T326, "Table functions", a <multiset value constructor> shall not contain a <table value constructor by query>.

This page intentionally left blank.

7 Query expressions

7.1 <row value constructor>

Function

Specify a value or list of values to be constructed into a row.

Format

```
<row value constructor> ::=
    <common value expression>
  | <boolean value expression>
  | <explicit row value constructor>

<explicit row value constructor> ::=
    <left paren> <row value constructor element> <comma>
    <row value constructor element list> <right paren>
  | ROW <left paren> <row value constructor element list> <right paren>
  | <row subquery>

<row value constructor element list> ::=
    <row value constructor element> [ { <comma> <row value constructor element> }... ]

<row value constructor element> ::= <value expression>

<contextually typed row value constructor> ::=
    <common value expression>
  | <boolean value expression>
  | <contextually typed value specification>
  | <left paren> <contextually typed row value specification> <right paren>
  | <left paren> <contextually typed row value constructor element> <comma>
    <contextually typed row value constructor element list> <right paren>
  | ROW <left paren> <contextually typed row value constructor element list> <right paren>

<contextually typed row value constructor element list> ::=
    <contextually typed row value constructor element>
    [ { <comma> <contextually typed row value constructor element> }... ]

<contextually typed row value constructor element> ::=
    <value expression>
  | <contextually typed value specification>

<row value constructor predicand> ::=
    <common value expression>
  | <boolean predicand>
  | <explicit row value constructor>
```

Syntax Rules

1) If a <row value constructor> is a <common value expression> or a <boolean value expression> *X*, then the <row value constructor> is equivalent to

```
ROW ( X )
```

2) If a <row value constructor predicand> is a <common value expression> or a <boolean predicand> *X*, then the <row value constructor predicand> is equivalent to

```
ROW ( X )
```

3) Let *ERVC* be an <explicit row value constructor>.

   Case:

   a) If *ERVC* simply contains a <row subquery>, then the declared type of *ERVC* is the declared type of that <row subquery>.

   b) Otherwise, the declared type of *ERVC* is a row type described by a sequence of (<field name>, <data type>) pairs, corresponding in order to each <row value constructor element> *X* simply contained in *ERVC*. The data type is the declared type of *X* and the <field name> is implementation-dependent.

4) If a <row value constructor> or <row value constructor predicand> *RVC* is an <explicit row value constructor> *ERVC*, then the declared type of *RVC* is the declared type of *ERVC*.

5) Let *CTRVC* be the <contextually typed row value constructor>.

   a) If *CTRVC* is a <common value expression>, <boolean value expression>, or <contextually typed value specification> *X*, then *CTRVC* is equivalent to:

```
ROW ( X )
```

   b) After the syntactic transformation specified in SR 5)a) has been performed, if necessary, the declared type of *CTRVC* is a row type described by a sequence of (<field name>, <data type>) pairs, corresponding in order to each <contextually typed row value constructor element> *X* simply contained in *CTRVC*. The <data type> is the declared type of *X* and the <field name> is implementation-dependent.

6) The degree of a <row value constructor>, <contextually typed row value constructor>, or <row value constructor predicand> is the degree of its declared type.

Access Rules

None.

General Rules

1) The value of a <null specification> is the null value.

2) The value of a <default specification> is determined according to the General Rules of Subclause 11.5, "<default clause>".

3) The value of an <empty specification> is an empty collection.

4) Case:

   a) If a <row value constructor>, <row value constructor predicand>, or <contextually typed row value constructor> immediately contains a <common value expression>, <boolean value expression>, or <contextually typed row value constructor element> *X*, then the result of the <row value constructor>, <row value constructor predicand>, or <contextually typed row value constructor> is a row containing a single column whose value is the value of *X*.

   b) If an <explicit row value constructor> is specified, then the result of the <row value constructor> or <row value constructor predicand> is a row of columns, the value of whose *i*-th column is the value of the *i*-th <row value constructor element> simply contained in the <explicit row value constructor>.

   c) If a <contextually typed row value constructor element list> is specified, then the result of the <contextually typed row value constructor> is a row of columns, the value of whose *i*-th column is the value of the *i*-th <contextually typed row value constructor element> in the <contextually typed row value constructor element list>.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain an <explicit row value constructor> that immediately contains ROW.

2) Without Feature T051, "Row types", conforming SQL language shall not contain a <contextually typed row value constructor> that immediately contains ROW.

3) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain an <explicit row value constructor> that is not simply contained in a <table value constructor> and that contains more than one <row value constructor element>.

4) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain an <explicit row value constructor> that is a <row subquery>.

5) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <row value constructor predicand> that immediately contains a <boolean predicand>.

6) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain a <contextually typed row value constructor> that is not simply contained in a <contextually typed table value constructor> and that contains more than one <row value constructor element>.

7) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain a <contextually typed row value constructor> that is a <row subquery>.

7.2 <row value expression>

Function

Specify a row value.

Format

```
<row value expression> ::=
    <row value special case>
  | <explicit row value constructor>

<table row value expression> ::=
    <row value special case>
  | <row value constructor>

<contextually typed row value expression> ::=
    <row value special case>
  | <contextually typed row value constructor>

<row value predicand> ::=
    <row value special case>
  | <row value constructor predicand>

<row value special case> ::= <nonparenthesized value expression primary>
```

Syntax Rules

1) The declared type of a <row value special case> shall be a row type.

2) The declared type of a <row value expression> is the declared type of the immediately contained <row value special case> or <explicit row value constructor>.

3) The declared type of a <table row value expression> is the declared type of the immediately contained <row value special case> or <row value constructor>.

4) The declared type of a <contextually typed row value expression> is the declared type of the immediately contained <row value special case> or <contextually typed row value constructor>. The declared type of a <row value predicand> is the declared type of the immediately contained <row value special case> or <row value constructor predicand>.

Access Rules

None.

General Rules

1) A <row value special case> specifies the row value denoted by the <nonparenthesized value expression primary>.

2) A <row value expression> specifies the row value denoted by the <row value special case> or <explicit row value constructor>.

3) A <table row value expression> specifies the row value denoted by the <row value special case> or <row value constructor>.

4) A <contextually typed row value expression> specifies the row value denoted by the <row value special case> or <contextually typed row value constructor>.

5) A <row value predicand> specifies the row value denoted by the <row value special case> or <row value constructor predicand>.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain a <row value special case>.

7.3 <table value constructor>

Function

Specify a set of <row value expression>s to be constructed into a table.

Format

```
<table value constructor> ::= VALUES <row value expression list>

<row value expression list> ::=
    <table row value expression> [ { <comma> <table row value expression> }... ]

<contextually typed table value constructor> ::=
    VALUES <contextually typed row value expression list>

<contextually typed row value expression list> ::=
    <contextually typed row value expression>
    [ { <comma> <contextually typed row value expression> }... ]
```

Syntax Rules

1) All <table row value expression>s immediately contained in a <row value expression list> shall be of the same degree.

2) All <contextually typed row value expression>s immediately contained in a <contextually typed row value expression list> shall be of the same degree.

3) A <table value constructor> or a <contextually typed table value constructor> is *possibly non-deterministic* if it generally contains a possibly non-deterministic <value expression>.

4) Let *TVC* be some <table value constructor> consisting of n <table row value expression>s or some <contextually typed table value constructor> consisting of n <contextually typed row value expression>s. Let RVE_i, 1 (one) $\leq i \leq n$, denote the i-th <table row value expression> or the i-th <contextually typed row value expression>. The row type of *TVC* is determined by applying Subclause 9.3, "Data types of results of aggregations", to the row types RVE_i, 1 (one) $\leq i \leq n$. The column names are implementation-dependent.

Access Rules

None.

General Rules

1) If the result of any <table row value expression> or <contextually typed row value expression> is the null value, then an exception condition is raised: *data exception — null row not permitted in table.*

2) The result *T* of a <table value constructor> or <contextually typed table value constructor> *TVC* is a table whose cardinality is the number of <table row value expression>s or the number of <contextually typed row value expression>s in *TVC*. If *R* is the result of *n* such expressions, then *R* occurs *n* times in *T*.

Conformance Rules

1) Without Feature F641, "Row and table constructors", in conforming SQL language, the <contextually typed row value expression list> of a <contextually typed table value constructor> shall contain exactly one <contextually typed row value constructor> *RVE*. *RVE* shall be of the form "(<contextually typed row value constructor element list>)".

2) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain a <table value constructor>.

7.4 <table expression>

Function

Specify a table or a grouped table.

Format

```
<table expression> ::=
    <from clause>
    [ <where clause> ]
    [ <group by clause> ]
    [ <having clause> ]
    [ <window clause> ]
```

Syntax Rules

1) The result of a <table expression> is a derived table whose row type *RT* is the row type of the result of the application of last of the immediately contained <from clause>, <where clause>, <group by clause>, or <having clause> specified in the <table expression>, together with the window structure descriptors defined by the <window clause>, if specified.

2) Let *C* be some column. Let *TE* be the <table expression>. *C* is an underlying column of *TE* if and only if *C* is an underlying column of some column reference contained in *TE*.

Access Rules

None.

General Rules

1) If all optional clauses are omitted, then the result of the <table expression> is the same as the result of the <from clause>. Otherwise, each specified clause is applied to the result of the previously specified clause and the result of the <table expression> is the result of the application of the last specified clause.

Conformance Rules

None.

7.5 <from clause>

Function

Specify a table derived from one or more tables.

Format

```
<from clause> ::= FROM <table reference list>

<table reference list> ::=
    <table reference> [ { <comma> <table reference> }... ]
```

Syntax Rules

1) Let *TRL* be the ordering of <table reference list>. No element TR_i in *TRL* shall contain an outer reference to an element TR_j, where $i \leq j$.

2) Case:

   a) If the <table reference list> immediately contains a single <table reference>, then the descriptor of the result of the <table reference list> is the same as the descriptor of the table identified by that <table reference>. The row type *RT* of the result of the <table reference list> is the row type of the table identified by the <table reference>.

   b) If the <table reference list> immediately contains more than one <table reference>, then the descriptors of the columns of the result of the <table reference list> are the descriptors of the columns of the tables identified by the <table reference>s, in the order in which the <table reference>s appear in the <table reference list> and in the order in which the columns are defined within each table. The row type *RT* of the result of the <table reference list> is determined by the sequence *SCD* of column descriptors of the result as follows:

      i) Let *n* be the number of column descriptors in *SCD*. *RT* has *n* fields.

      ii) For *i* ranging from 1 (one) to *n*, the field name of the *i*-th field descriptor in *RT* is the column name included in the *i*-th column descriptor in *SCD*.

      iii) For *i* ranging from 1 (one) to *n*, the data type descriptor of the *i*-th field descriptor in *RT* is

      Case:

         1) If the *i*-th descriptor in *SCD* includes a domain name *DN*, then the data type descriptor included in the descriptor of the domain identified by *DN*.

         2) Otherwise, the data type descriptor included in the *i*-th column descriptor in *SCD*.

3) The descriptor of the result of the <from clause> is the same as the descriptor of the result of the <table reference list>.

Access Rules

None.

General Rules

1) Let *TRLR* be the result of *TRL*.

   Case:

   a) If *TRL* simply contains a single <table reference> *TR*, then *TRLR* is the result of *TR*.

   b) If *TRL* simply contains n <table reference>s, where $n > 1$, then let *TRLP* be the <table reference list> formed by taking the first n–1 elements of *TRL* in order, let *TRLL* be the last element of *TRL*, and let *TRLPR* be the result of *TRLP*. For every row R_i, 1 (one) $\leq i \leq n$, in *TRLPR*, let $TRLLR_i$ be the corresponding evaluation of *TRLL* under all outer references contained in *TRLL*. Let $SUBR_i$ be the table containing every row formed by concatenating R_i with some row of $TRLLR_i$. Every row *RR* in $SUBR_i$ is a row in *TRLR*, and the number of occurrences of *RR* in *TRLR* is the sum of the numbers of occurrences of *RR* in every occurrence of $SUBR_i$.

   The result of the <table reference list> is *TRLR* with the columns reordered according to the ordering of the descriptors of the columns of the <table reference list>.

2) The result of the <from clause> is *TRLR*.

Conformance Rules

None.

7.6 <table reference>

Function

Reference a table.

Format

```
<table reference> ::=
    <table factor>
  | <joined table>

<table factor> ::= <table primary> [ <sample clause> ]

<sample clause> ::=
    TABLESAMPLE <sample method> <left paren> <sample percentage> <right paren>
    [ <repeatable clause> ]

<sample method> ::=
    BERNOULLI
  | SYSTEM

<repeatable clause> ::= REPEATABLE <left paren> <repeat argument> <right paren>

<sample percentage> ::= <numeric value expression>

<repeat argument> ::= <numeric value expression>

<table primary> ::=
    <table or query name> [ [ AS ] <correlation name>
    [ <left paren> <derived column list> <right paren> ] ]
  | <derived table> [ AS ] <correlation name>
    [ <left paren> <derived column list> <right paren> ]
  | <lateral derived table> [ AS ] <correlation name>
    [ <left paren> <derived column list> <right paren> ]
  | <collection derived table> [ AS ] <correlation name>
    [ <left paren> <derived column list> <right paren> ]
  | <table function derived table> [ AS ] <correlation name>
    [ <left paren> <derived column list> <right paren> ]
  | <only spec> [ [ AS ] <correlation name>
    [ <left paren> <derived column list> <right paren> ] ]
  | <left paren> <joined table> <right paren>

<only spec> ::= ONLY <left paren> <table or query name> <right paren>

<lateral derived table> ::= LATERAL <table subquery>

<collection derived table> ::=
    UNNEST <left paren> <collection value expression> <right paren>
    [ WITH ORDINALITY ]

<table function derived table> ::=
    TABLE <left paren> <collection value expression> <right paren>

<derived table> ::= <table subquery>
```

```
<table or query name> ::=
    <table name>
  | <transition table name>
  | <query name>

<derived column list> ::= <column name list>

<column name list> ::= <column name> [ { <comma> <column name> }... ]
```

Syntax Rules

1) The declared type of <repeat argument> shall be an exact numeric type with scale 0 (zero).

2) If a <table primary> *TP* simply contains a <table function derived table> *TFDT*, then:

   a) The <collection value expression> immediately contained in *TFDT* shall be a <routine invocation>.

   b) Let *CN* be the <correlation name> simply contained in *TP*.

   c) Let *CVE* be the <collection value expression> simply contained in *TP*.

   d) Case:

      i) If *TP* specifies a <derived column list> *DCL*, then let *TFDCL* be
      ```
      ( DCL )
      ```

      ii) Otherwise, let *TFDCL* be a zero-length string.

   e) *TP* is equivalent to the <table primary>
   ```
   UNNEST ( CVE ) AS CN TFDCL
   ```

3) If a <table primary> *TP* simply contains a <collection derived table> *CDT*, then let *CVE* be the <collection value expression> simply contained in *CDT*, let *CN* be the <correlation name> simply contained in *TP*, and let *TEMP* be an <identifier> that is not equivalent to *CN* nor to any other <identifier> contained in *TP*. Let *ET* be the element type of the declared type of *CVE*.

   a) Case:

      i) If the declared type of *CVE* is a multiset, then WITH ORDINALITY shall not be specified. Let *IMDC* be the implementation-defined maximum cardinality of an array whose declared element type is *ET*. Let *C* be
      ```
      ( CAST (CVE AS ET ARRAY[IMDC]) )
      ```

      ii) Otherwise, let *C* be *CVE*.

   b) Let *N1* and *N2* be two <column name>s that are not equivalent to one another nor to *CN*, *TEMP*, or any other <identifier> contained in *TP*.

   c) Let *RECQP* be:
   ```
   WITH RECURSIVE TEMP(N1, N2) AS
     (   SELECT C[1] AS N1, 1 AS N2
         FROM (VALUES(1)) AS CN
         WHERE 0 < CARDINALITY(C)
   ```

```
    UNION
      SELECT C[N2+1] AS N1, N2+1 AS N2
      FROM TEMP
      WHERE N2 < CARDINALITY(C)
)
```

d) Case:

i) If *TP* specifies a <derived column list> *DCL*, then:

1) Case:

A) If *CDT* specifies WITH ORDINALITY, then

Case:

I) If *ET* is a row type, then let *DET* be the degree of *ET*. *DCL* shall contain *DET*+1 (one) <column name>s.

II) Otherwise, *DCL* shall contain 2 <column name>s.

B) Otherwise,

Case:

I) If *ET* is a row type, then let *DET* be the degree of *ET*. *DCL* shall contain *DET* <column name>s.

II) Otherwise, *DCL* shall contain 1 (one) <column name>.

2) Let *PDCLP* be

```
( DCL )
```

ii) Otherwise,

Case:

1) If *ET* is a row type, then:

A) Let *DET* be the degree of *ET*.

B) Let FN_i, 1 (one) ≤ *i* ≤ *DET*, be the name of the *i*-th field in *ET*.

C) Case:

I) If *CDT* specifies WITH ORDINALITY, then let *PDCLP* be:

```
( FN1, FN2, ..., FNDET, N2 )
```

II) Otherwise, let *PDCLP* be:

```
( FN1, FN2, ..., FNDET )
```

2) Otherwise, let *PDCLP* be a zero-length string.

e) Case:

i) If *CDT* specifies WITH ORDINALITY, then

Case:

1) If *ET* is a row type, then let *ELDT* be:

```
LATERAL ( RECQP SELECT N1.*, N2
                FROM TEMP ) AS CN PDCLP
```

2) Otherwise, let *ELDT* be:

```
LATERAL ( RECQP SELECT *
                FROM TEMP ) AS CN PDCLP
```

ii) Otherwise,

Case:

1) If *ET* is a row type, then let *ELDT* be:

```
LATERAL ( RECQP SELECT N1.*
                FROM TEMP ) AS CN PDCLP
```

2) Otherwise, let *ELDT* be:

```
LATERAL ( RECQP SELECT N1
                FROM TEMP ) AS CN PDCLP
```

f) *TP* is equivalent to the <table primary> *ELDT*.

4) If a <table factor> *TF* simply contains a <correlation name>, then let *RV* be that <correlation name>; otherwise, let *RV* be the <table or query name> simply contained in *TF*. *RV* is a range variable. *RV* is *exposed* by *TF*.

NOTE 124 — "range variable" is defined in Subclause 4.14.6, "Operations involving tables".

5) If a <table factor> *TF* is contained in a <from clause> *FC* with no intervening <query expression>, then the *scope clause SC* of *TF* is the <select statement: single row> or innermost <query specification> that contains *FC*. The scope of the range variable of *TF* is the <select list>, <where clause>, <group by clause>, <having clause>, and <window clause> of *SC*, together with every <lateral derived table> that is simply contained in *FC* and is preceded by *TF*, and every <collection derived table> that is simply contained in *FC* and is preceded by *TF*, and the <join condition> of all <joined table>s contained in *SC* that contain *TF*. If *SC* is the <query specification> that is the <query expression body> of a simple table query *STQ*, then the scope of the range variable of *TF* also includes the <order by clause> of *STQ*.

NOTE 125 — "simple table query" is defined in Subclause 14.1, "<declare cursor>".

6) If a <table factor> *TF* is simply contained in a <merge statement> *MS*, then the *scope clause SC* of *TF* is *MS*. The scope of the range variable of *TF* is the <search condition>, <set clause list>, and <merge insert value list> of *SC*.

7) Let *RV* be the range variable that is exposed by a <table factor> *TF*. Let *RV1* be the range variable that is exposed by a <table factor> *TF1* that has the same scope clause as *TF*.

Case:

a) If *RV* is a <table name>, then

Case:

i) If *RV1* is a <table name>, then *RV1* shall not be equivalent to *RV*.

ii) Otherwise, *RV1* shall not be equivalent to the <qualified identifier> of *RV*.

b) Otherwise,

Case:

i) If *RV1* is a <table name>, then the <qualified identifier> of *RV1* shall not be equivalent to *RV*.

ii) Otherwise, *RV1* shall not be equivalent to *RV*.

8) A <table or query name> simply contained in a <table factor> *TF* has a scope clause and scope defined by *TF* if and only if the <table or query name> is exposed by *TF*.

9) If a <table primary> *TP* simply contains <table or query name> *TOQN*, then

Case:

a) If *TOQN* is an <identifier> that is equivalent to a <query name> *QN*, then let *WLE* be the <with list element> simply contained in the <query expression> that simply contains *TP* such that the <query name> *QN1* simply contained in *WLE* is equivalent to *QN* and *QN1* is the innermost query name in scope. Let the *table specified by the <query name>* be the result of *WLE*.

NOTE 126 — "query name in scope" is defined in Subclause 7.13, "<query expression>".

b) If *TOQN* is an <identifier> that is equivalent to a <transition table name> that is in scope, then let the *table specified by the <transition table name>* be the table identified by *TOQN*.

NOTE 127 — The scope of a <transition table name> is defined in Subclause 11.39, "<trigger definition>".

c) Otherwise, let the *table specified by the <table name>* be the table identified by the <table name> simply contained in *TP*.

NOTE 128 — The preceding cases disambiguate whether *TOQN* is interpreted as a <query name>, <transition table name>, or <table name>.

10) If a <table primary> *TP* simply contains <only spec> *OS* and the table identified by the <table or query name> *TN* is not a typed table, then *OS* is equivalent to *TN*.

11) No <column name> shall be specified more than once in a <derived column list>.

12) If a <derived column list> is specified in a <table primary> *TP*, then the number of <column name>s in the <derived column list> shall be the same as the degree of the table specified by the <derived table>, the <lateral derived table>, or the <table or query name> simply contained in *TP*, and the name of the *i*-th column of that <derived table> or <lateral derived table> or the effective name of the *i*-th column of that <table or query name> is the *i*-th <column name> in that <derived column list>.

13) The row type of a <lateral derived table> is the row type of the simply contained <query expression>.

14) Case:

a) If no <derived column list> is specified in a <table primary> *TP*, then the row type *RT* of *TP* is the row type of its simply contained <table or query name>, <derived table>, <lateral derived table>, or <joined table>.

b) Otherwise, the row type *RT* of *TP* is described by a sequence of (<field name>, <data type>) pairs, where the <field name> in the *i*-th pair is the *i*-th <column name> in the <derived column list> and the <data type> in the *i*-th pair is the declared type of the *i*-th column of the <derived table>, <joined table>, <lateral derived table>, or of the table identified by the <table or query name> simply contained in *TP*.

15) A <derived table> or <lateral derived table> is an *updatable derived table* if and only if the <query expression> simply contained in the <derived table> or <lateral derived table> is updatable.

16) A <derived table> or <lateral derived table> is an *insertable-into derived table* if and only if the <query expression> simply contained in the <derived table> or <lateral derived table> is insertable-into.

17) A <collection derived table> is not updatable.

18) If a <table reference> *TR* immediately contains a <table factor> *TF*, then

Case:

a) If *TF* simply contains a <table name> that identifies a base table, then every column of the table identified by *TF* is called an *updatable column* of *TR*.

b) If *TF* simply contains a <table name> that identifies a view, then every updatable column of the view identified by *TF* is called an *updatable column* of *TR*.

c) If *TF* simply contains a <derived table> or <lateral derived table>, then every updatable column of the table identified by the <query expression> simply contained in <derived table> or <lateral derived table> is called an *updatable column* of *TR*.

19) If a <table reference> *TR* immediately contains a <table factor> and the <table or query name> simply contained in *TR* immediately contains a <table name> *TN*, then let *T* be the table identified by *TN*. The schema identified by the explicit or implicit qualifier of *TN* shall include the descriptor of *T*.

20) A <table name> is *possibly non-deterministic* if the table identified by the <table name> is a viewed table, and the original <query expression> in the view descriptor identified by the <table name> is possibly non-deterministic.

21) A <query name> is *possibly non-deterministic* if the <query expression> identified by the <query name> is possibly non-deterministic.

22) A <derived table> or <lateral derived table> is *possibly non-deterministic* if the simply contained <query expression> is possibly non-deterministic.

23) A <table primary> is *possibly non-deterministic* if the simply contained <table name>, <query name>, <derived table>, <lateral derived table>, or <joined table> is possibly non-deterministic.

24) A <table reference> is *possibly non-deterministic* if the simply contained <table primary> or <joined table> is possibly non-deterministic or if <sample clause> is specified.

Access Rules

1) If a <table primary> *TP* simply contains a <table or query name> that simply contains a <table name> *TN*, then:

   a) Let *T* be the table identified by *TN*.

   b) Case:

      i) If *TN* is contained in a <search condition> immediately contained in an <assertion definition>or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on at least one column of *T*.

      ii) Otherwise:

         1) Case:

            A) If *TP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on at least one column of *T*.

            B) Otherwise, the current privileges shall include SELECT on at least one column of *T*.

         2) If *TP* simply contains <only spec> and *TN* identifies a typed table, then

            Case:

            A) If *TP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

            B) Otherwise, the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

      NOTE 129 — "applicable privileges" and "current privileges" are defined in Subclause 12.3, "<privileges>".

General Rules

1) If a <table primary> *TP* simply contains a <table or query name> *TOQN*, then

   Case:

   a) If *TOQN* simply contains a <query name> *QN*, then the result of *TP* is the table specified by *QN*.

   b) If *TOQN* simply contains a <transition table name> *TTN*, then the result of *TP* is the table specified by *TTN*.

      NOTE 130 — The table identified by a <transition table name> is a transition table as defined by the General Rules of Subclause 14.16, "Effect of deleting rows from base tables", Subclause 14.19, "Effect of inserting tables into base tables", or Subclause 14.22, "Effect of replacing rows in base tables", as appropriate.

   c) Otherwise, let *T* be the table specified by the <table name> simply contained in *TP*.

      Case:

i) If ONLY is specified, then the result of *TP* is a table that consists of every row in *T*, except those rows that have a subrow in a proper subtable of *T*.

ii) Otherwise, the result of *TP* is a table that consists of every row of *T*.

2) If a <derived table> or <lateral derived table> *LDT* simply containing <query expression> *QE* is specified, then the result of *LDT* is the result of *QE*.

3) Let *TP* be the <table primary> immediately contained in a <table factor> *TF*. Let *RT* be the result of *TP*.

Case:

a) If <sample clause> is specified, then:

i) Let *N* be the number of rows in *RT* and let *S* be the value of <sample percentage>.

ii) If *S* is the null value or if $S < 0$ (zero) or if $S > 100$, then an exception condition is raised: *data exception — invalid sample size.*

iii) If <repeatable clause> is specified, then let *RPT* be the value of <repeat argument>. If *RPT* is the null value, then an exception condition is raised: *data exception — invalid repeat argument in a sample clause.*

iv) Case:

1) If <sample method> specifies BERNOULLI, then the result of *TF* is a table containing approximately (*N***S*/100) rows of *RT*. The probability of a row of *RT* being included in result of *TF* is *S*/100. Further, whether a given row of *RT* is included in result of *TF* is independent of whether other rows of *RT* are included in result of *TF*.

2) Otherwise, result of *TF* is a table containing approximately (*N***S*/100) rows of *RT*. The probability of a row of *RT* being included in result of *TF* is *S*/100.

b) Otherwise, result of *TF* is *RT*.

4) The result of a <table reference> *TR* is the result of immediately contained <table factor> or <joined table>.

5) Let *RV* be the range variable that is exposed by a <table factor> *TF*. The table associated with *RV* is the result of *TF*.

Conformance Rules

1) Without Feature S091, "Basic array support", or Feature S271, "Basic multiset support", conforming SQL language shall not contain a <collection derived table>.

2) Without Feature T491, "LATERAL derived table", conforming SQL language shall not contain a <lateral derived table>.

3) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", conforming SQL language shall not contain a <query name>.

4) Without Feature S111, "ONLY in query expressions", conforming SQL language shall not contain a <table reference> that contains an <only spec>.

5) Without Feature F591, "Derived tables", conforming SQL language shall not contain a <derived table>.

6) Without Feature T326, "Table functions", conforming SQL language shall not contain a <table function derived table>.

7) Without Feature T613, "Sampling", conforming SQL language shall not contain a <sample clause>.

8) Without Feature T211, "Basic trigger capability", conforming SQL language shall not contain a <transition table name>.

7.7 <joined table>

Function

Specify a table derived from a Cartesian product, inner join, or outer join.

Format

```
<joined table> ::=
    <cross join>
  | <qualified join>
  | <natural join>

<cross join> ::=
    <table reference> CROSS JOIN <table factor>

<qualified join> ::=
    <table reference> [ <join type> ] JOIN <table reference> <join specification>

<natural join> ::=
    <table reference> NATURAL [ <join type> ] JOIN <table factor>

<join specification> ::=
    <join condition>
  | <named columns join>

<join condition> ::= ON <search condition>

<named columns join> ::= USING <left paren> <join column list> <right paren>

<join type> ::=
    INNER
  | <outer join type> [ OUTER ]

<outer join type> ::=
    LEFT
  | RIGHT
  | FULL

<join column list> ::= <column name list>
```

Syntax Rules

1) Let TR_1 be the first <table reference>, and let TR_2 be the <table reference> or <table factor> that is the second operand of the <joined table>. Let RT_1 and RT_2 be the row types of TR_1 and TR_2, respectively. Let *TA* and *TB* be the range variables of TR_1 and TR_2, respectively. Let *CP* be:

```
SELECT *
FROM TR1, TR2
```

2) If TR_2 contains a <lateral derived table> containing an outer reference that references TR_1, then <join type> shall not contain RIGHT or FULL.

3) If a <qualified join> or <natural join> is specified and a <join type> is not specified, then INNER is implicit.

4) If a <qualified join> containing a <join condition> is specified and a <value expression> directly contained in the <search condition> is a <set function specification>, then the <joined table> shall be contained in a <having clause> or <select list>, the <set function specification> shall contain an aggregated argument *AA* that contains an outer reference, and every column reference contained in *AA* shall be an outer reference.

NOTE 131 — "outer reference" is defined in Subclause 6.7, "<column reference>".

5) The <search condition> shall not contain a <window function> without an intervening <subquery>.

6) If neither NATURAL is specified nor a <join specification> immediately containing a <named columns join> is specified, then the descriptors of the columns of the result of the <joined table> are the same as the descriptors of the columns of *CP*, with the possible exception of the nullability characteristics of the columns.

7) If NATURAL is specified or if a <join specification> immediately containing a <named columns join> is specified, then:

   a) If NATURAL is specified, then let *common column name* be a <field name> that is equivalent to the <field name> of exactly one field of RT_1 and the <field name> of exactly one field of RT_2. RT_1 shall not have any duplicate common column names and RT_2 shall not have any duplicate common column names. Let *corresponding join columns* refer to all fields of RT_1 and RT_2 that have common column names, if any.

   b) If a <named columns join> is specified, then every <column name> in the <join column list> shall be equivalent to the <field name> of exactly one field of RT_1 and the <field name> of exactly one field of RT_2. Let *common column name* be the name of such a column. Let *corresponding join columns* refer to the columns identified in the <join column list>.

   c) Let C_1 and C_2 be a pair of corresponding join columns of RT_1 and RT_2, respectively. C_1 and C_2 shall be comparable. C_1 and C_2 are operands of an equality operation, and the Syntax Rules of Subclause 9.9, "Equality operations", apply.

   d) If there is at least one corresponding join column, then let *SLCC* be a <select list> of <derived column>s of the form

      ```
      COALESCE ( TA.C, TB.C ) AS C
      ```

      for every column *C* that is a corresponding join column, taken in order of their ordinal positions in RT_1.

   e) If RT_1 contains at least one field that is not a corresponding join column, then let SLT_1 be a <select list> of <derived column>s of the form

      ```
      TA.C
      ```

      for every field *C* of RT_1 that is not a corresponding join column, taken in order of their ordinal positions in RT_1.

f) If RT_2 contains at least one field that is not a corresponding join column, then let SLT_2 be a <select list> of <derived column>s of the form

```
TB.C
```

for every field C of RT_2 that is not a corresponding join column, taken in order of their ordinal positions in RT_2.

g) Let the <select list> *SL* be defined as

Case:

i) If all of the fields of RT_1 and RT_2 are corresponding join columns, then let *SL* be "*SLCC*".

ii) If RT_1 contains no corresponding join columns and RT_2 contains no corresponding join columns, then let *SL* be "SLT_1, SLT_2".

iii) If RT_1 contains no fields other than corresponding join columns, then let *SL* be "*SLCC*, SLT_2".

iv) If RT_2 contains no fields other than corresponding join columns, then let *SL* be "*SLCC*, SLT_1".

v) Otherwise, let *SL* be "*SLCC*, SLT_1, SLT_2".

The descriptors of the columns of the result of the <joined table>, with the possible exception of the nullability characteristics of the columns, are the same as the descriptors of the columns of the result of

```
SELECT SL FROM TR1, TR2
```

8) A <joined table> is *possibly non-deterministic* if at least one of the following conditions is true:

a) Either TR_1 or TR_2 is possibly non-deterministic.

b) A <join condition> that generally contains a possibly non-deterministic <value expression>, possibly non-deterministic <query specification>, or possibly non-deterministic <query expression> is specified.

c) NATURAL is specified, or a <join specification> immediately containing a <named columns join> is specified, and there is a common column name *CCN* such that the declared types of the two corresponding join columns identified by *CCN* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

9) The declared type of the rows of the <joined table> is the row type *RT* defined by the sequence of (<field name>, <data type>) pairs indicated by the sequence of column descriptors of the <joined table> taken in order.

10) For every column *CR* of the result of the <joined table> that corresponds to a field C_1 of RT_1 that is not a corresponding join column, *CR* is *possibly nullable* if any of the following conditions are true:

a) RIGHT or FULL is specified.

b) INNER, LEFT, or CROSS JOIN is specified or implicit and C_1 is possibly nullable.

11) For every column *CR* of the result of the <joined table> that corresponds to a field C_2 of RT_2 that is not a corresponding join column, *CR* is *possibly nullable* if any of the following conditions are true:

a) LEFT or FULL is specified.

b) INNER, RIGHT, or CROSS JOIN is specified or implicit and C_2 is possibly nullable.

12) For every column *CR* of the result of the <joined table> that corresponds to a corresponding join column C_1 of RT_1 and a corresponding join column C_2 of RT_2, *CR* is *possibly nullable* if any of the following conditions are true:

a) LEFT or FULL is specified and C_1 is possibly nullable, or

b) RIGHT or FULL is specified and C_2 is possibly nullable.

Access Rules

None.

General Rules

1) Let T_1 be the result of evaluating TR_1.

2) Case:

a) If a <cross join> is specified, then let *T* be *CP*.

b) If a <join condition> is specified, then let *SC* be the <search condition> and let *T* be

```
CP
WHERE SC
```

c) If NATURAL is specified or <named columns join> is specified, then

Case:

i) If there are corresponding join columns, then let *N* be the number of coresponding join columns and let $CJCN_i$, 1 (one) ≤ i ≤ N, be the field name of the i-th corresponding join column, and let *T* be

```
CP
WHERE TA.CJCN1 = TB.CJCN1
  AND ...
  AND TA.CJCNN = TB.CJCNN
```

ii) Otherwise, let *T* be *CP*.

3) Let *TR* be the result of evaluating *T*.

4) Let P_1 be the collection of rows of T_1 for which there exists in *TR* some row that is a subrow of some row R_1 of T_1.

5) Let U_1 be those rows of T_1 that are not in P_1.

6) Let D_1 and D_2 be the degrees of TR_1 and TR_2, respectively. Let X_1 be U_1 extended on the right with D_2 columns containing the null value.

7) Let XN_1 be an effective distinct name for X_1. Let *TN* be an effective name for *T*.

8) If RIGHT or FULL is specified, then:

   a) Let T_2 be the result of evaluating TR_2.

   b) Let P_2 be the collection of rows of T_2 for which there exists in *TR* some row that is a subrow of some row R_1 of T_1.

   c) Let U_2 be those rows of T_2 that are not in P_2.

   d) Let X_2 be U_2 extended on the left with D_1 columns containing the null value.

   e) Let XN_2 be an effective distinct name for X_2.

9) Case:

   a) If INNER or <cross join> is specified, then let *S* be *TR*.

   b) If LEFT is specified, then let *S* be the result of:

```
SELECT * FROM TN
UNION ALL
SELECT * FROM XN1
```

   c) If RIGHT is specified, then let *S* be the result of:

```
SELECT * FROM TN
UNION ALL
SELECT * FROM XN2
```

   d) If FULL is specified, then let *S* be the result of:

```
SELECT * FROM TN
UNION ALL
SELECT * FROM XN1
UNION ALL
SELECT * FROM XN2
```

10) Let *SN* be an effective name of *S*.

    Case:

    a) If NATURAL is specified or a <named columns join> is specified, then:

       i) Let CS_i be a name for the *i*-th column of *S*. Column CS_i of *S* corresponds to the *i*-th field of RT_1 if *i* is less than or equal to D_1. Column CS_j of *S* corresponds to the $(j–D_1)$-th field of RT_2 for *j* greater than D_1.

ii) If there is at least one corresponding join column, then let *SLCC* be a <select list> of derived columns of the form

```
COALESCE (CSi, CSj)
```

for every pair of columns CS_i and CS_j, where CS_i and CS_j correspond to fields of RT_1 and RT_2 that are a pair of corresponding join columns.

iii) If RT_1 contains one or more fields that are not corresponding join columns, then let SLT_1 be a <select list> of the form:

```
CSi
```

for every column CS_i of *S* that corresponds to a field of RT_1 that is not a corresponding join column, taken in order of their ordinal position in *S*.

iv) If RT_2 contains one or more fields that are not corresponding join columns, then let SLT_2 be a <select list> of the form:

```
CSj
```

for every column CS_j of *S* that corresponds to a field of RT_2 that is not a corresponding join column, taken in order of their ordinal position in *S*.

v) Let the <select list> *SL* be defined as

Case:

1) If all the fields of RT_1 and RT_2 are corresponding join columns, then let *SL* be

```
SLCC
```

2) If RT_1 contains no corresponding join columns and RT_2 contains no corresponding join columns, then let *SL* be

```
SLT1, SLT2
```

3) If RT_1 contains no fields other than corresponding join columns, then let *SL* be

```
SLCC, SLT2
```

4) If RT_2 contains no fields other than corresponding join columns, then let *SL* be

```
SLCC, SLT1
```

5) Otherwise, let *SL* be

```
SLCC, SLT1, SLT2
```

vi) The result of the <joined table> is the result of:

```
SELECT SL FROM SN
```

b) Otherwise, the result of the <joined table> is *S*.

Conformance Rules

1) Without Feature F401, "Extended joined table", conforming SQL language shall not contain a <cross join>.

2) Without Feature F401, "Extended joined table", conforming SQL language shall not contain a <natural join>.

3) Without Feature F401, "Extended joined table", conforming SQL language shall not contain FULL.

4) Without Feature F402, "Named column joins for LOBs, arrays, and multisets", conforming SQL language shall not contain a <joined table> that simply contains either <natural join> or <named columns join> in which, if *C* is a corresponding join column, the declared type of *C* is LOB-ordered, array-ordered, or multiset-ordered.

NOTE 132 — If *C* is a corresponding join column, then the Conformance Rules of Subclause 9.9, "Equality operations", also apply.

7.8 <where clause>

Function

Specify a table derived by the application of a <search condition> to the result of the preceding <from clause>.

Format

```
<where clause> ::= WHERE <search condition>
```

Syntax Rules

1) If a <value expression> directly contained in the <search condition> is a <set function specification>, then the <where clause> shall be contained in a <having clause> or <select list>, the <set function specification> shall contain a column reference, and every column reference contained in an aggregated argument of the <set function specification> shall be an outer reference.

   NOTE 133 — *outer reference* is defined in Subclause 6.7, "<column reference>".

2) The <search condition> shall not contain a <window function> without an intervening <subquery>.

Access Rules

None.

General Rules

1) Let *T* be the result of the preceding <from clause>.

2) The <search condition> is applied to each row of *T*. The result of the <where clause> is a table of those rows of *T* for which the result of the <search condition> is *True*.

3) Each <subquery> that is directly contained in the <search condition> is effectively executed for each row of *T* and the results used in the application of the <search condition> to the given row of *T*.

Conformance Rules

1) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <value expression> directly contained in a <where clause> that contains a <column reference> that references a <derived column> that generally contains a <set function specification> without an intervening <routine invocation>.

7.9 <group by clause>

Function

Specify a grouped table derived by the application of the <group by clause> to the result of the previously specified clause.

Format

```
<group by clause> ::=
    GROUP BY [ <set quantifier> ] <grouping element list>

<grouping element list> ::=
    <grouping element> [ { <comma> <grouping element> }... ]

<grouping element> ::=
    <ordinary grouping set>
  | <rollup list>
  | <cube list>
  | <grouping sets specification>
  | <empty grouping set>

<ordinary grouping set> ::=
    <grouping column reference>
  | <left paren> <grouping column reference list> <right paren>

<grouping column reference> ::=
    <column reference> [ <collate clause> ]

<grouping column reference list> ::=
    <grouping column reference> [ { <comma> <grouping column reference> }... ]

<rollup list> ::=
    ROLLUP <left paren> <ordinary grouping set list> <right paren>

<ordinary grouping set list> ::=
    <ordinary grouping set> [ { <comma> <ordinary grouping set> }... ]

<cube list> ::=
    CUBE <left paren> <ordinary grouping set list> <right paren>

<grouping sets specification> ::=
    GROUPING SETS <left paren> <grouping set list> <right paren>

<grouping set list> ::=
    <grouping set> [ { <comma> <grouping set> }... ]

<grouping set> ::=
    <ordinary grouping set>
  | <rollup list>
  | <cube list>
  | <grouping sets specification>
  | <empty grouping set>
```

```
<empty grouping set> ::= <left paren> <right paren>
```

Syntax Rules

1) Each <grouping column reference> shall unambiguously reference a column of the table resulting from the <from clause>. A column referenced in a <group by clause> is a *grouping column*.

   NOTE 134 — "Column reference" is defined in Subclause 6.7, "<column reference>".

2) Each <grouping column reference> is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

3) For every <grouping column reference> *GC*,

   Case:

   a) If <collate clause> is specified, then let *CS* be the collation identified by <collation name>. The declared type of the column reference shall be character string. The declared type of *GC* is that of its column reference, except that *CS* is the declared type collation and the collation derivation is *explicit*.

   b) Otherwise, the declared type of *GC* is the declared type of its column reference.

4) Let *QS* be the <query specification> that simply contains the <group by clause>, and let *SL*, *FC*, *WC*, *GBC*, and *HC* be the <select list>, the <from clause>, the <where clause> if any, the <group by clause>, and the <having clause> if any, respectively, that are simply contained in *QS*.

5) Let *QSSQ* be the explicit or implicit <set quantifier> immediately contained in *QS*.

6) Let *GBSQ* be the <set quantifier> immediately contained in <group by clause>, if any; otherwise, let *GBSQ* be ALL.

7) Let *SL1* be obtained from *SL* by replacing every <asterisk> and <asterisked identifier chain> using the syntactic transformations in the Syntax Rules of Subclause 7.12, "<query specification>".

8) A <group by clause> is *primitive* if it does not contain a <rollup list>, <cube list>, <grouping sets specification>, or <grouping column reference list>, and does not contain both a <grouping column reference> and an <empty grouping set>.

9) A <group by clause> is *simple* if it does not contain a <rollup list>, <cube list> or <grouping sets specification>.

10) If *GBC* is a simple <group by clause> that is not primitive, then *GBC* is transformed into a primitive <group by clause> as follows:

    a) Let *NSGB* be the number of <grouping column reference>s contained in *GBC*.

    b) Case:

       i) If *NSGB* is 0 (zero), then *GBC* is replaced by

          ```
          GROUP BY ()
          ```

       ii) Otherwise:

1) Let $SGCR_1$, ... $SGCR_{NSGB}$ be an enumeration of the <grouping column reference>s contained in *GBC*.

2) *GBC* is replaced by

```
GROUP BY SGCR1, ... SGCRNSGB
```

NOTE 135 — That is, a simple <group by clause> that is not primitive may be transformed into a primitive <group by clause> by deleting all parentheses, and deleting extra <comma>s as necessary for correct syntax. If there are no grouping columns at all (for example, `GROUP BY (), ()`), this is transformed to the canonical form `GROUP BY ()`.

11) If *GBC* is a primitive <group by clause>, then let *SLNEW* and *HCNEW* be obtained from *SL1* and *HC*, respectively, by replacing every <grouping operation> by the exact numeric literal 0 (zero). *QS* is equivalent to:

```
SELECT QSSQ SLNEW FC WC GBC HCNEW
```

12) If *OGSL* is an <ordinary grouping set list>, then the *concatenation* of *OGSL* is defined as follows:

a) Let *NGCR* be the number of <grouping column reference>s simply contained in *OGSL* and let GCR_j, 1 (one) $\leq j \leq$ *NGCR*, be an enumeration of those <grouping column reference>s, in order from left to right.

b) The concatenation of *OGSL* is the <ordinary grouping set list>

```
GCR1, ..., GCRNGCR
```

NOTE 136 — Thus, the concatenation of *OGSL* may be formed by erasing all parentheses. For example, the concatenation of "(A, B), (C, D)" is "A, B, C, D".

13) If *RL* is a <rollup list>, then let OGS_i range over the *n* <ordinary grouping set>s contained in *RL*.

a) For each *i* between 1 (one) and *n*, let $COGS_i$ be the concatenation of the <ordinary grouping set list>

```
ORG1, ORG2, ..., ORGi
```

b) *RL* is equivalent to:

```
GROUPING SETS (
    ( COGSn ),
    ( COGSn-1 ),
    ( COGSn-2 ),
    ...
    ( COGS1 ),
    ( ) )
```

NOTE 137 — The result of the transform is to replace *RL* with a <grouping sets specification> that contains a <grouping set> for every initial sublist of the <ordinary grouping set list> of the <rollup list>, obtained by dropping <ordinary grouping set>s from the right, one by one, and concatenating each <ordinary grouping set list> so obtained. The <empty grouping set> is regarded as the shortest such initial sublist. For example, "ROLLUP ((A, B), (C, D))" is equivalent to "GROUPING SETS ((A, B, C, D), (A, B), ())".

14) If *CL* is a <cube list>, then let OGS_i range over the *n* <ordinary grouping set>s contained in *CL*. *CL* is transformed as follows:

a) Let $M = 2^n - 1$ (one).

b) For each i between 1 (one) and M:

i) Let BSL_i be the binary number consisting of n bits (binary digits) whose value is i.

ii) For each j between 1 (one) and n, let $B_{i,j}$ be the j-th bit, counting from left to right, in BSL_i.

iii) For each j between 1 (one) and n, let $GSLCR_{i,j}$ be

Case:

1) If $B_{i,j}$ is 0 (zero), then the zero-length string.

2) If $B_{i,j}$ is 1 (one) and $B_{i,k}$ is 0 (zero) for all $k < j$, then OGS_j.

3) Otherwise, <comma> followed by OGS_j.

iv) Let GSL_i be the concatenation of the <ordinary grouping set list>

$GSLCR_{i,1}$ $GSLCR_{i,2}$ ... $GSLCR_{i,n}$

c) *CL* is equivalent to

GROUPING SETS ((GSL_M), (GSL_{M-1}), ..., (GSL_1), ( ))

NOTE 138 — The result of the transform is to replace *CL* with a <grouping sets specification> that contains a <grouping set> for all possible subsets of the set of <ordinary grouping set>s in the <ordinary grouping set list> of the <cube list>, including <empty grouping set> as the empty subset with no <ordinary grouping set>s.

For example, `CUBE (A, B, C)` is equivalent to:

```
GROUPING SETS ( /* BSL_i */
  (A, B, C),    /* 111 */
  (A, B   ),    /* 110 */
  (A,    C),    /* 101 */
  (A      ),    /* 100 */
  (   B, C),    /* 011 */
  (   B   ),    /* 010 */
  (      C),    /* 001 */
  (       )
)
```

As another example, `CUBE ((A, B), (C, D))` is equivalent to:

```
GROUPING SETS ( /* BSL_i */
  (A, B, C, D), /* 11 */
  (A, B      ), /* 10 */
  (      C, D), /* 01 */
  (           )
)
```

15) If <grouping sets specification> *GSSA* simply contains another <grouping sets specification> *GSSB*, then *GSSA* is transformed as follows:

a) Let *NA* be the number of <grouping set>s simply contained in *GSSA*, and let *NB* be the number of <grouping set>s simply contained in *GSSB*.

b) Let GSA_i be an enumeration of the <grouping set>s simply contained in *GSSA*, for 1 (one) $\le i \le NA$.

c) Let GSB_i be an enumeration of the <grouping set>s simply contained in *GSSB*, 1 (one) $\le i \le NB$.

d) Let *k* be the value such that $GSSB = GSA_k$.

e) *GSSA* is equivalent to

```
GROUPING SETS (
    GSA1, GSA2, ... GSAk-1,
    GSB1, ... , GSBNB,
    GSAk+1, ..., GSANA )
```

NOTE 139 — Thus, the nested <grouping sets specification> is removed by simply "promoting" each of its <grouping set>s to be a <grouping set> of the encompassing <grouping sets specification>.

16) If *CGB* is a <group by clause> that is not simple, then *CGB* is transformed as follows:

a) The preceding Syntax Rules are applied repeatedly to eliminate any <grouping sets specification> that is nested in another <grouping sets specification>, as well as any <rollup list> and any <cube list>.

NOTE 140 — As a result, *CGB* is a list of two or more <grouping set>s, each of which is an <ordinary grouping set>, an <empty grouping set>, or a <grouping sets specification> that contains only <ordinary grouping set>s and <empty grouping set>s. There are no remaining <rollup list>s, <cube list>s, or nested <grouping sets specification>s.

b) Any <grouping element> *GS* that is an <ordinary grouping set> or an <empty grouping set> is replaced by the <grouping sets specification>

```
GROUPING SETS ( GS )
```

NOTE 141 — As a result, *CGB* is a list of two or more <grouping sets specification>s.

c) Let *GSSX* and *GSSY* be the first two <grouping sets specification>s in *CGB*. *CGB* is transformed by replacing "*GSSX* <comma> *GSSY*" as follows:

i) Let *NX* be the number of <grouping set>s in *GSSX* and let *NY* be the number of <grouping set>s in *GSSY*.

ii) Let GSX_i, 1 (one) $\le i \le NX$, be the <grouping set>s contained in *GSSX*, and let GSY_i, 1 (one) $\le i \le NY$, be the <grouping set>s contained in *GSSY*.

iii) Let *MX*(*i*) be the number of <grouping column reference>s in GSX_i, and let *MY*(*i*) be the number of <grouping column reference>s in GSY_i.

NOTE 142 — If GSX_i is <empty grouping set>, then *MX*(*i*) is 0 (zero); and similarly for GSY_i.

iv) Let $GCRX_{i,j}$, 1 (one) $\le j \le MX(i)$ be the <grouping column reference>s contained in GSX_i, and let $GCRY_{i,j}$, 1 (one) $\le j \le MY(i)$ be the <grouping column reference>s contained in GSY_i.

NOTE 143 — If GSX_i is <empty grouping set>, then there are no $GCRX_{i,j}$; and similarly for GSY_i.

v) For each *a* between 1 (one) and *NX* and each *b* between 1 (one) and *NY*, let $GST_{a,b}$ be

($GCRX_{a,1}$, ..., $GCRX_{a,MX(a)}$, $GCRY_{b,1}$, ..., $GCRY_{b,MY(b)}$)

that is, an <ordinary grouping set> consisting of $GCRA_{a,j}$ for all *j* between 1 (one) and *MX*(*a*), followed by $GCRY_{b,j}$ for all *j* between 1 (one) and *MY*(*b*).

vi) *CGB* is transformed by replacing "*GSSX* <comma> *GSSY*" with

```
GROUPING SETS (
  GST1,1, ..., GST1,NY,
  GST2,1, ..., GST2,NY,
  ...
  GSTNX,1 , ..., GSTNX,NY
)
```

NOTE 144 — Thus each <ordinary grouping set> in *GSSA* is "concatenated" with each <ordinary grouping set> in *GSSB*. For example,

```
GROUP BY GROUPING SETS ((A, B), (C)),
         GROUPING SETS ((X, Y), ())
```

is transformed to

```
GROUP BY GROUPING SETS ((A, B, X, Y), (A, B),
         (C, X, Y), (C))
```

d) The previous subrule of this Syntax Rule is applied repeatedly until *CGB* consists of a single <grouping sets specification>.

17) If <grouping element list> consists of a single <grouping sets specification> *GSS* that contains only <ordinary grouping set>s or <empty grouping set>s, then:

a) Let *m* be the number of <grouping set>s contained in *GSS*.

b) Let GS_i, $1 \le i \le m$, range over the <grouping set>s contained in *GSS*.

c) Let *p* be the number of distinct <column reference>s that are contained in *GSS*.

d) Let *PC* be an ordered list of these <column reference>s ordered according to their left-to-right occurrence in the list.

e) Let PC_k, $1 \le k \le p$, be the *k*-th <column reference> in *PC*.

f) Let $DTPC_k$ be the declared type of the column identified by PC_k.

g) Let *NDC* be the number of <derived column>s simply contained in *SL1*.

h) Let DC_q, $1 \le q \le NDC$, be an enumeration of the <derived column>s simply contained in SL1, in order from left to right.

i) Let DCN_q be the column name of DC_q, 1 (one) $\le q \le NDC$.

j) Let VE_q, 1 (one) ≤ q ≤ NDC, be the <value expression> simply contained in DC_q.

k) Let XN_k, 1 (one) ≤ k ≤ p, YN_k, 1 (one) ≤ k ≤ p, and ZN_q, 1 (one) ≤ q ≤ NDC, be implementation-dependent column names that are all distinct from one another.

l) Let *SL2* be the <select list>:

```
PC1 AS XN1, GROUPING (PC1) AS YN1,
...,
PCp AS XNp, GROUPING (PCp) AS YNp,
VE1 AS XN1, ..., VENDC AS ZNNDC
```

m) For each GS_i:

i) If GS_i is an <empty grouping set>, then let $n(i)$ be 0 (zero). If GS_i is a <grouping column reference>, then let $n(i)$ be 1 (one). Otherwise, let $n(i)$ be the number of <grouping column reference>s contained in the <grouping column reference list>.

ii) Let $GCR_{i,j}$, 1 ≤ j ≤ $n(i)$, range over the <grouping column reference>s contained in GS_i.

iii) Case:

1) If GS_i is an <ordinary grouping set>, then

A) Transform *SL2* to obtain *SL3*, and transform *HC* to obtain *HC3*, as follows:

For every PC_k, if there is no j such that $PC_k = GCR_{i,j}$, then make the following replacements in *SL2* and *HC*:

I) Replace each <grouping operation> in *SL2* and *HC* that contains a <column reference> that references PC_k by the <literal> 1 (one).

II) Replace each <column reference> in *SL2* and *HC* that references PC_k by

```
CAST ( NULL AS DTPCk )
```

B) Transform *SL3* to obtain *SLNEW*, and transform *HC3* to obtain *HCNEW* by replacing each <grouping operation> that remains in *SL3* and *HC3* by the <literal> 0 (zero).

NOTE 145 — Thus the value of a <grouping operation> is 0 (zero) if the grouping column referenced by the <grouping operation> is among the $GCR_{i,j}$ and 1(one) if it is not.

C) Let $GSSQL_i$ be:

```
SELECT QSSQ SLNEW
FC
WC
GROUP BY GCRi,1, ..., GCRi,n(i)
HCNEW
```

2) If GS_i is an <empty grouping set>, then

A) Transform *SL2* to obtain *SLNEW*, and transform *HC* to obtain *HCNEW*, as follows:

For every k, $1 \le k \le p$:

I) Replace each <grouping operation> in *SL2* and *HC* that contains a <column reference> that references PC_k by the <literal> 1 (one).

II) Replace each <column reference> in *SL2* and *HC* that references PC_k by

```
CAST ( NULL AS DTPC_k )
```

B) Let $GSSQL_i$ be

```
SELECT QSSQ SLNEW
FC
WC
GROUP BY ()
HCNEW
```

n) Let *GU* be:

```
GSSQL_1
  UNION GBSQ
GSSQL_2
  UNION GBSQ
...
  UNION GBSQ
GSSQL_m
```

o) *QS* is equivalent to

```
SELECT QSSQ ZN_1 AS DC_1, ..., ZN_NDC AS DC_NDC
FROM ( GU )
```

Access Rules

None.

General Rules

NOTE 146 — As a result of the syntactic transformations specified in the Syntax Rules of this Subclause, only primitive <group by clause>s are left to consider.

1) If no <where clause> is specified, then let *T* be the result of the preceding <from clause>; otherwise, let *T* be the result of the preceding <where clause>.

2) Case:

a) If there are no grouping columns, then the result of the <group by clause> is the grouped table consisting of *T* as its only group.

b) Otherwise, the result of the <group by clause> is a partitioning of the rows of *T* into the minimum number of groups such that, for each grouping column of each group, no two values of that grouping column are distinct. If the declared type of a grouping column is a user-defined type and the comparison of that column results in *Unknown* for two rows of *T*, then the assignment of those rows to groups in the result of the <group by clause> is implementation-dependent.

3) When a <search condition> or <value expression> is applied to a group, a reference *CR* to a column that is functionally dependent on the grouping columns is understood as follows.

Case:

a) If *CR* is a group-invariant column reference, then it is a reference to the common value in that column of the rows in that group. If the most specific type of the column is character, datetime with time zone, or a user-defined type, then the value is an implementation-dependent value that is not distinct from the value of the column in each row of the group.

b) Otherwise, *CR* is a within-group-varying column reference, and as such, it is a reference to the value of the column in each row of a given group determined by the grouping columns, to be used to construct the argument source of a <set function specification>.

Conformance Rules

1) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <rollup list>.

2) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <cube list>.

3) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <grouping sets specification>.

4) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain an <empty grouping set>.

5) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain an <ordinary grouping set> that contains a <grouping column reference list>.

6) Without Feature T432, "Nested and concatenated GROUPING SETS", conforming SQL language shall not contain a <grouping set list> that contains a <grouping sets specification>.

7) Without Feature T432, "Nested and concatenated GROUPING SETS", conforming SQL language shall not contain a <group by clause> that simply contains a <grouping sets specification> *GSS* where *GSS* is not the only <grouping element> simply contained in the <group by clause>.

NOTE 147 — The Conformance Rules of Subclause 9.10, "Grouping operations", also apply.

8) Without Feature T434, "GROUP BY DISTINCT", conforming SQL language shall not contain a <group by clause> that simply contains a <set quantifier>.

7.10 <having clause>

Function

Specify a grouped table derived by the elimination of groups that do not satisfy a <search condition>.

Format

```
<having clause> ::= HAVING <search condition>
```

Syntax Rules

1) Let *HC* be the <having clause>. Let *TE* be the <table expression> that immediately contains *HC*. If *TE* does not immediately contain a <group by clause>, then "GROUP BY ()" is implicit. Let *T* be the descriptor of the table defined by the <group by clause> *GBC* immediately contained in *TE* and let *R* be the result of *GBC*.

2) Let *G* be the set consisting of every column referenced by a <column reference> contained in *GBC*.

3) Each column reference directly contained in the <search condition> shall be one of the following:

   a) An unambiguous reference to a column that is functionally dependent on *G*.

   b) An outer reference.

   NOTE 148 — See also the Syntax Rules of Subclause 6.7, "<column reference>".

4) The <search condition> shall not contain a <window function> without an intervening <subquery>.

5) The row type of the result of the <having clause> is the row type *RT* of *T*.

Access Rules

None.

General Rules

1) The <search condition> is applied to each group of *R*. The result of the <having clause> is a grouped table of those groups of *R* for which the result of the <search condition> is *True*.

2) Each <subquery> that is directly contained in the <search condition> is effectively evaluated for each group of *R* and the result used in the application of the <search condition> to the given group of *R*.

Conformance Rules

1) Without Feature T301, "Functional dependencies", in conforming SQL language, each column reference directly contained in the <search condition> shall be one of the following:

a) An unambiguous reference to a grouping column of *T*.

b) An outer reference.

2) Without Feature T301, "Functional dependencies", in conforming SQL language, each column reference contained in a <subquery> in the <search condition> that references a column of *T* shall be one of the following:

a) An unambiguous reference to a grouping column of *T*.

b) Contained in an aggregated argument of a <set function specification>.

7.11 <window clause>

Function

Specify one or more window definitions.

Format

```
<window clause> ::= WINDOW <window definition list>

<window definition list> ::=
    <window definition> [ { <comma> <window definition> }... ]

<window definition> ::= <new window name> AS <window specification>

<new window name> ::= <window name>

<window specification> ::=
    <left paren> <window specification details> <right paren>

<window specification details> ::=
    [ <existing window name> ]
    [ <window partition clause> ]
    [ <window order clause> ]
    [ <window frame clause> ]

<existing window name> ::= <window name>

<window partition clause> ::=
    PARTITION BY <window partition column reference list>

<window partition column reference list> ::=
    <window partition column reference>
    [ { <comma> <window partition column reference> }... ]

<window partition column reference> ::=
    <column reference> [ <collate clause> ]

<window order clause> ::=
    ORDER BY <sort specification list>

<window frame clause> ::=
    <window frame units> <window frame extent>
    [ <window frame exclusion> ]

<window frame units> ::=
    ROWS
  | RANGE

<window frame extent> ::=

    <window frame start>
  | <window frame between>

<window frame start> ::=
```

```
    UNBOUNDED PRECEDING
  | <window frame preceding>
  | CURRENT ROW

<window frame preceding> ::= <unsigned value specification> PRECEDING

<window frame between> ::= BETWEEN <window frame bound 1> AND <window frame bound 2>

<window frame bound 1> ::= <window frame bound>

<window frame bound 2> ::= <window frame bound>

<window frame bound> ::=
    <window frame start>
  | UNBOUNDED FOLLOWING
  | <window frame following>

<window frame following> ::= <unsigned value specification> FOLLOWING

<window frame exclusion> ::=
    EXCLUDE CURRENT ROW
  | EXCLUDE GROUP
  | EXCLUDE TIES
  | EXCLUDE NO OTHERS
```

Syntax Rules

1) Let *TE* be the <table expression> that immediately contains the <window clause>.

2) <new window name> *NWN1* shall not be contained in the scope of another <new window name> *NWN2* such that *NWN1* and *NWN2* are equivalent.

3) Let *WDEF* be a <window definition>.

4) Each <column reference> contained in the <window partition clause> or <window order clause> of *WDEF* shall unambiguously reference a column of the derived table *T* that is the result of *TE*. A column referenced in a <window partition clause> is a *partitioning column*. Each partitioning column is an operand of a grouping operation, and the Syntax Rules of Subclause 9.10, "Grouping operations", apply.

   NOTE 149 — If *T* is a grouped table, then the <column reference>s contained in <window partition clause> or <window order clause> shall reference columns of the grouped table obtained by performing the syntactic transformation in Subclause 7.12, "<query specification>".

5) For every <window partition column reference> *PC*,

   Case:

   a) If <collate clause> is specified, then let *CS* be the collation identified by <collation name>. The declared type of the column reference shall be character string. The declared type of *PC* is that of its column reference, except that *CS* is the declared type collation and the collation derivation is *explicit*.

   b) Otherwise, the declared type of *PC* is the declared type of its column reference.

6) If *T* is a grouped table, then let *G* be the set of grouping columns of *T*. Each column reference contained in <window clause> that references a column of *T* shall reference a column that is functionally dependent on *G* or be contained in an aggregated argument of a <set function specification>.

7) A <window clause> shall not contain a <window function> without an intervening <subquery>.

8) If *WDEF* specifies <window frame between>, then:

   a) <window frame bound 1> shall not specify UNBOUNDED FOLLOWING.

   b) <window frame bound 2> shall not specify UNBOUNDED PRECEDING.

   c) If <window frame bound 1> specifies CURRENT ROW, then <window frame bound 2> shall not specify <window frame preceding>.

   d) If <window frame bound 1> specifies <window frame following>, then <window frame bound 2> shall not specify <window frame preceding> or CURRENT ROW.

9) If *WDEF* specifies <window frame extent>, and does not specify <window frame between>, then let *WAGS* be the <window frame start>. The <window frame extent> is equivalent to

```
BETWEEN WAGS AND CURRENT ROW
```

10) If *WDEF* specifies an <existing window name> *EWN*, then:

   a) *WDEF* shall be within the scope of a <window name> that is equivalent to <existing window name>.

   b) Let *WDX* be the window structure descriptor identified by *EWN*.

   c) *WDEF* shall not specify <window partition clause>.

   d) If *WDX* has a window ordering clause, then *WDEF* shall not specify <window order clause>.

   e) *WDX* shall not have a window framing clause.

11) If *WDEF*'s <window frame clause> specifies <window frame preceding> or <window frame following>, then let *UVS* be the <unsigned value specification> simply contained in the <window frame preceding> or <window frame following>.

   Case:

   a) If RANGE is specified, then *WDEF*'s <window order clause> shall contain a single <sort key> *SK*. The declared type of *SK* shall be numeric, datetime, or interval. The declared type of *UVS* shall be numeric if the declared type of *SK* is numeric; otherwise, it shall be an interval type that may be added to or subtracted from the declared type of *SK* according to the Syntax Rules of Subclause 6.30, "<datetime value expression>", and Subclause 6.32, "<interval value expression>", in this part of ISO/IEC 9075.

   b) If ROWS is specified, then the declared type of *UVS* shall be exact numeric with scale 0 (zero).

12) The scope of the <new window name> simply contained in *WDEF* consists of any <window definition>s that follow *WDEF* in the <window clause>, together with the <select list> of the <query specification> or <select statement: single row> that simply contains the <window clause>. If the <window clause> is simply contained in a <query specification> that is the <query expression body> of a <declare cursor> that is a simple table query, then the scope of <new window name> also includes the <order by clause> of the <declare cursor>.

13) Two window structure descriptors *WD1* and *WD2* are *order-equivalent* if all of the following conditions are met:

a) Let $WPCR1_i$, 1 (one) $\le i \le N1$, and $WPCR2_i$, 1 (one) $\le i \le N2$, be enumerations of the <window partition column reference>s contained in the window partitioning clauses of *WD1* and *WD2*, respectively, in order from left to right. $N1 = N2$, and, for all i, $WPCR1_i$ and $WPCR2_i$ are equivalent column references.

b) Let $SS1_i$, 1 (one) $\le i \le M1$, and $SS2_i$, 1 (one) $\le i \le M2$, be enumerations of the <sort specification>s contained in the window ordering clauses of *WD1* and *WD2*, respectively, in order from left to right. $M1 = M2$, and, for all i, $SS1_i$ and $SS2_i$ contain <sort key>s that are equivalent column references, specify or imply the same <ordering specification>, specify or imply the same <collate clause>, if any, and specify or imply the same <null ordering>.

Access Rules

None.

General Rules

1) Let *TE* be the <table expression> that simply contains the <window clause>. Let *SL* be the <select list> of the <query specification> or <select statement: single row> that immediately contains *TE*.

Case:

a) If *SL* does not simply contain a <window function>, then the <window clause> is disregarded, and the result of *TE* is the result of the last <from clause>, <where clause>, <group by clause> or <having clause> of *TE*.

b) Otherwise, let *RTE* be the result of the last <from clause> or <where clause> simply contained in *TE*.

NOTE 150 — Although it is permissible to have a <group by clause> or a <having clause> with a <window clause>, if there are any <window function>s, then the <group by clause> and <having clause> are removed by a syntactic transformation in Subclause 7.12, "<query specification>", and so are not considered here.

i) A window structure descriptor *WDESC* is created for each <window definition> *WDEF*, as follows:

1) *WDESC*'s window name is the <new window name> simply contained in *WDEF*.

2) If <existing window name> is specified, then let *EWN* be the <existing window name> simply contained in *WDEF* and let *WDX* be the window structure descriptor identified by *EWN*.

3) If <existing window name> is specified and the window ordering clause of *WDX* is present, then the ordering window name of *WDESC* is *EWN*; otherwise, there is no ordering window name.

4) Case:

A) If *WDEF* simply contains <window partition clause> *WDEFWPC*, then *WDESC*'s window partitioning clause is *WDEFWPC*.

B) If <existing window name> is specified, then *WDESC*'s window partitioning clause is the window partitioning clause of *WDX*.

C) Otherwise, *WDESC* has no window partitioning.

5) Case:

A) If *WDEF* simply contains <window order clause> *WDEFWOC*, then *WDESC*'s window ordering clause is *WDEFWOC*.

B) If <existing window name> is specified, then *WDESC*'s window ordering clause is the window ordering clause of *WDX*.

C) Otherwise, *WDESC* has no window ordering.

6) If *WDEF* simply contains <window frame clause> *WDEFWFC*, then *WDESC*'s window framing clause is *WDEFWFC*; otherwise, *WDESC* has no windows framing.

ii) The result of <window clause> is *RTE*, together with the window structure descriptors defined by the <window clause>.

2) Let *WD* be a window structure descriptor.

3) *WD* defines, for each row *R* of *RTE*, the *window partition* of *R* under *WD*, consisting of the collection of rows of *RTE* that are not distinct from *R* in the window partitioning columns of *WD*. If *WD* has no window partitioning clause, then the window partition of *R* is the entire result *RTE*.

4) *WD* also defines the window ordering of the rows of each window partition defined by *WD*, according to the General Rules of Subclause 10.10, "<sort specification list>", using the <sort specification list> simply contained in *WD*'s window ordering clause. If *WD* has no window ordering clause, then the window ordering is implementation-dependent, and all rows are peers. Although the window ordering of peer rows within a window partition is implementation-dependent, the window ordering shall be the same for all window structure descriptors that are order-equivalent. It shall also be the same for any pair of windows *W1* and *W2* such that *W1* is the ordering window for *W2*.

5) *WD* also defines for each row *R* of *RTE* the window frame *WF* of *R*, consisting of a collection of rows. *WF* is defined as follows.

Case:

a) If *WD* has no window framing clause, then

Case:

i) If the window ordering clause of *WD* is not present, then *WF* is the window partition of *R*.

ii) Otherwise, *WF* consists of all rows of the partition of *R* that precede *R* or are peers of *R* in the window ordering of the window partition defined by the window ordering clause.

b) Otherwise, let *WF* initially be the window partition of *R* defined by *WD*. Let *WFC* be the window framing clause of *WD*. Let *WFB1* be the <window frame bound 1> and let *WFB2* be the <window frame bound 2> contained in *WFC*.

i) If RANGE is specified, then:

1) In the following subrules, when performing addition or subtraction to combine a datetime and a year-month interval, if the result would raise the exception condition *data exception — datetime field overflow* because the <primary datetime field> DAY is not valid for the computer value of the <primary datetime field>s YEAR and MONTH, then the <primary

datetime field> DAY is set to the last day that is valid for the <primary datetime field>s YEAR and MONTH, and no exception condition is raised.

2) Case:

NOTE 151 — In the following subrules, if *WFB1* specifies UNBOUNDED PRECEDING, then no rows are removed from *WF* by this step. *WFB1* may not be UNBOUNDED FOLLOWING.

A) If *WFB1* specifies <window frame preceding>, then let *V1P* be the value of the <unsigned value specification>.

Case:

I) If *V1P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function.*

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

a) If NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

b) Case:

i) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK+V1P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

ii) Otherwise, let *BOUND* be the value *VSK–V1P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

B) If *WFB1* specifies CURRENT ROW, then remove from *WF* all rows that are not peers of the current row and that precede the current row in the window ordering defined by *WD*.

C) If *WFB1* specifies <window frame following>, then let *V1F* be the value of the <unsigned value specification>.

Case:

I) If *V1F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function.*

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

   a) If NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

   b) Case:

      i) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK–V1F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

      ii) Otherwise, let *BOUND* be the value *VSK+V1F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

3) Case:

NOTE 152 — In the following subrules, if *WFB2* specifies UNBOUNDED FOLLOWING, then no rows are removed from *WF* by this step. *WFB2* may not be UNBOUNDED PRECEDING.

A) If *WFB2* specifies <window frame preceding>, then let *V2P* be the value of the <unsigned value specification>.

Case:

I) If *V2P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

   a) If NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

   b) Case:

      i) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK+V2P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

ii) Otherwise, let *BOUND* be the value *VSK–V2P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

B) If *WFB2* specifies CURRENT ROW, then remove from *WF* all rows following the current row in the ordering defined by *WD* that are not peers of the current row.

C) If *WFB2* specifies <window frame following>, then let *V2F* be the value of the <unsigned value specification>.

Case:

I) If *V2F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

a) If NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

b) Case:

i) If the <ordering specification> contained in the <window order clause> specifies DESC, then let *BOUND* be the value *VSK–V2F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

ii) Otherwise, let *BOUND* be the value *VSK+V2F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

ii) If ROWS is specified, then:

1) Case:

NOTE 153 — In the following subrules, if *WFB1* specifies UNBOUNDED PRECEDING, then no rows are removed from *WF* by this step. *WFB1* may not be UNBOUNDED FOLLOWING.

A) If *WFB1* specifies <window frame preceding>, then let *V1P* be the value of the <unsigned value specification>.

Case:

I) If *V1P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that are more than *V1P* rows preceding the current row in the window ordering defined by *WD*.

B) If *WFB1* specifies CURRENT ROW, then remove from *WF* all rows that precede the current row in the window ordering defined by *WD*.

NOTE 154 — This step removes any peers of the current row that precede it in the implementation-dependent window ordering.

C) If *WFB1* specifies <window frame following>, then let *V1F* be the value of the <unsigned value specification>.

Case:

I) If *V1F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that precede the current row and all rows that are less than *V1F* rows following the current row in the window ordering defined by *WD*.

NOTE 155 — If *V1F* is zero, then the current row is not removed from *WF* by this step; otherwise, the current row is removed from *WF*.

2) Case:

NOTE 156 — In the following subrules, if *WFB2* specifies UNBOUNDED FOLLOWING, then no rows are removed from *WF* by this step. *WFB2* may not be UNBOUNDED PRECEDING.

A) If *WFB2* specifies <window frame preceding>, then let *V2P* be the value of the <unsigned value specification>.

Case:

I) If *V2P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that follow the current row and all rows that are less than *V2P* rows preceding the current row in the window ordering defined by *WD*.

NOTE 157 — If *V2P* is zero, then the current row is not removed from *WF* by this step; otherwise, the current row is removed from *WF*.

B) If *WFB2* specifies CURRENT ROW, then remove from *WF* all rows that follow the current row in the window ordering defined by *WD*.

NOTE 158 — This step removes any peers of the current row that follow it in the implementation-dependent window ordering.

C) If *WFB2* specifies <window frame following>, then let *V2F* be the value of the <unsigned value specification>.

Case:

I) If *V2F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that are more than *V2F* rows following the current row in the window ordering defined by *WD*.

iii) If <window frame exclusion> *WFE* is specified, then

Case:

1) If EXCLUDE CURRENT ROW is specified and the current row is still a member of *WF*, then remove the current row from *WF*.

2) If EXCLUDE GROUP is specified, then remove the current row and any peers of the current row from *WF*.

3) If EXCLUDE TIES is specified, then remove any rows other than the current row that are peers of the current row from *WF*.

NOTE 159 — If the current row is already removed from *WF*, then it remains removed from *WF*.

NOTE 160 — If EXCLUDE NO OTHERS is specified, then no additional rows are removed from *WF* by this Rule.

Conformance Rules

1) Without Feature T611, "Elementary OLAP operations", conforming SQL language shall not contain a <window specification>.

2) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window clause>.

3) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain an <existing window name>.

4) Without Feature T301, "Functional dependencies", in conforming SQL language, if *T* is a grouped table, then each column reference contained in <window clause> that references a column of *T* shall be a reference to a grouping column of *T* or be contained in an aggregated argument of a <set function specification>.

5) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window frame exclusion>.

NOTE 161 — The Conformance Rules of Subclause 9.10, "Grouping operations", also apply.

7.12 <query specification>

Function

Specify a table derived from the result of a <table expression>.

Format

```
<query specification> ::=
    SELECT [ <set quantifier> ] <select list> <table expression>

<select list> ::=
    <asterisk>
  | <select sublist> [ { <comma> <select sublist> }... ]

<select sublist> ::=
    <derived column>
  | <qualified asterisk>

<qualified asterisk> ::=
    <asterisked identifier chain> <period> <asterisk>
  | <all fields reference>

<asterisked identifier chain> ::=
    <asterisked identifier> [ { <period> <asterisked identifier> }... ]

<asterisked identifier> ::= <identifier>

<derived column> ::= <value expression> [ <as clause> ]

<as clause> ::= [ AS ] <column name>

<all fields reference> ::=
    <value expression primary> <period> <asterisk>
    [ AS <left paren> <all fields column name list> <right paren> ]

<all fields column name list> ::= <column name list>
```

Syntax Rules

1) Let *T* be the result of the <table expression>.

2) Let *TQS* be the table that is the result of a <query specification>.

3) Case:

   a) If the <select list> "*" is simply contained in a <subquery> that is immediately contained in an <exists predicate>, then the <select list> is equivalent to a <value expression> that is an arbitrary <literal>.

   b) Otherwise, the <select list> "*" is equivalent to a <value expression> sequence in which each <value expression> is a column reference that references a column of *T* and each column of *T* is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within *T*.

4) The degree of the table specified by a <query specification> is equal to the cardinality of the <select list>.

5) If a <set quantifier> DISTINCT is specified, then each column of *T* is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

6) The ambiguous case of an <all fields reference> whose <value expression primary> takes the form of an <asterisked identifier chain> shall be analyzed first as an <asterisked identifier chain> to resolve the ambiguity.

7) If <asterisked identifier chain> is specified, then:

   a) Let *IC* be an <asterisked identifier chain>.

   b) Let *N* be the number of <asterisked identifier>s immediately contained in *IC*.

   c) Let I_i, 1 (one) ≤ i ≤ N, be the <asterisked identifier>s immediately contained in *IC*, in order from left to right.

   d) Let PIC_1 be I_1. For each *J* between 2 and *N*, let PIC_J be $PIC_{J-1}.I_J$. PIC_J is called the *J*-th *partial identifier chain* of *IC*.

   e) Let *M* be the minimum of *N* and 3.

   f) For at most one *J* between 1 and *M*, PIC_J is called the *basis* of *IC*, and *J* is called the *basis length* of *IC*. The *referent* of the basis is a table *T*, a column *C* of a table, or an SQL parameter *SP*. The *basis* and *basis scope* of *IC* are defined in terms of a *candidate basis*, according to the following rules:

      i) If *IC* is contained in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_1, then PIC_1 is a candidate basis of *IC*, and the scope of PIC_1 is the scope of *SP*.

      ii) If $N = 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

      iii) If $N > 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes a refinable SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

      iv) If $N = 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. If the table associated with *EN* has a column *C* of row type whose <identifier> is equivalent to I_2, then PIC_2 is a candidate basis of *IC* and the scope of PIC_2 is the scope of *EN*.

      v) If $N > 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. If the table associated with *EN* has a column *C* of row type or structured type whose <identifier> is equivalent to I_2, then PIC_2 is a candidate basis of *IC* and the scope of PIC_2 is the scope of *EN*.

vi) If $N \leq 3$ and PIC_N is equivalent to an exposed <table or query name> that is in scope, then let *EN* be the exposed <table or query name> that is equivalent to PIC_N and has the innermost scope. PIC_N is a candidate basis of *IC*, and the scope of PIC_N is the scope of *EN*.

vii) There shall be exactly one candidate basis *CB* with innermost scope. The basis of *IC* is *CB*. The basis scope is the scope of *CB*.

g) Case:

i) If the basis is a <table or query name> or <correlation name>, then let *TQ* be the table associated with the basis. The <select sublist> is equivalent to a <value expression> sequence in which each <value expression> is a column reference *CR* that references a column of *TQ* that is not a common column of a <joined table>. Each column of *TQ* that is not a referenced common column shall be referenced exactly once. The columns shall be referenced in the ascending sequence of their ordinal positions within *TQ*.

ii) Otherwise let *BL* be the length of the basis of *IC*.

Case:

1) If $BL = N$, then the <select sublist> `IC.*` is equivalent to `(IC).*`.

2) Otherwise, the <select sublist> `IC.*` is equivalent to:

$$(PIC_{BL}) . I_{BL+1} . \ldots . I_N . *$$

NOTE 162 — The equivalent syntax in either case will be analyzed as <all fields reference> ::= <value expression primary> <period> <asterisk>

8) The data type of the <value expression primary> *VEP* specified in an <all fields reference> *AFR* shall be some row type *VER*. Let *n* be the degree of *VER*. Let F_1, ..., F_N be the field names of *VER*.

Case:

a) If <all fields column name list> *AFCNL* is specified, then the number of <column name>s simply contained in *AFCNL* shall be *n*. Let $AFCN_i$, 1 (one) $\leq i \leq n$, be these <column name>s in order from left to right. *AFR* is equivalent to

$$VEP . F_1 \text{ AS } AFCN_1, \ldots, VEP . F_n \text{ AS } AFCN_n)$$

b) Otherwise, *AFR* is equivalent to:

$$VEP . F_1, \ldots, VEP . F_n$$

9) Let *C* be some column. Let *QS* be the <query specification>. Let DC_i, for *i* ranging from 1 (one) to the number of <derived column>s inclusively, be the *i*-th <derived column> simply contained in the <select list> of *QS*. For all *i*, *C* is an underlying column of DC_i, and of any column reference that identifies DC_i, if and only if *C* is an underlying column of the <value expression> of DC_i, or *C* is an underlying column of the <table expression> immediately contained in *QS*.

10) Each column reference contained in a <window function> shall unambiguously reference a column of *T*.

11) If both of the following two conditions are satisfied, then *QS* is a *grouped, windowed query*:

a) *T* is a grouped table.

b) Some <derived column> simply contained in *QS* simply contains a <window function>.

12) A grouped, windowed query *GWQ* is transformed to an equivalent <query specification> as follows:

a) If *GWQ* contains an <in-line window specification>, then apply the syntactic transformation specified in Subclause 6.10, "<window function>".

b) If the <select list> of *GWQ* contains <asterisk> or <qualified asterisk>, then apply the syntactic transformations specified in Subclause 7.12, "<query specification>".

c) Let *GWQ2* be the result of the preceding transformations, if any.

d) Let *SL*, *FC*, *WC*, *GBC*, *HC*, and *WIC* be the <select list>, <from clause>, <where clause>, <group by clause>, <having clause>, and <window clause>, respectively, of *GWQ2*. If any of <where clause>, <group by clause>, or <having clause>, are missing, then let *WC*, *GBC*, and *HC*, respectively, be a zero-length string. Let *SQ* be the <set quantifier> immediately contained in the <query specification> of *GWQ2*, if any; otherwise, let *SQ* be a zero-length string.

NOTE 163 — *GWQ2* can not lack a <window clause>, since the syntactic transformation of Subclause 6.10, "<window function>", will create one if there is not one in *GWQ* already.

e) Let *N1* be the number of <set function specification>s simply contained in *GWQ2*.

f) Let SFS_i, 1 (one) ≤ *i* ≤ *N1*, be an enumeration of the <set function specification>s simply contained in *GWQ2*.

g) Let $SFSI_i$, 1 (one) ≤ *i* ≤ *N1*, be a list of <identifier>s that are distinct from each other and distinct from all <identifier>s contained in *GWQ2*.

h) If *N1* = 0 (zero), then let *SFSL* be a zero-length string; otherwise, let *SFSL* be:

SFS_1 AS $SFSI_1$, SFS_2 AS $SFSI_2$, ..., SFS_{N1} AS $SFSI_{N1}$

i) Let *HCNEW* be obtained from *HC* by replacing each <set function specification> SFS_i by the corresponding <identifier> $SFSI_i$.

j) Let *N2* be the number of <column reference>s that are contained in *SL* or *WIC* without an intervening <subquery> or <set function specification>.

k) Let CR_j, 1 (one) ≤ *j* ≤ *N2*, be an enumeration of the <column reference>s that are contained in *SL* or *WIC* without an intervening <subquery> or <set function specification>.

l) Let CRI_j, 1 (one) ≤ *j* ≤ *N2*, be a list of <identifier>s that are distinct from each other, distinct from all identifiers in *GWQ2*, and distinct from all $SFSI_i$.

m) If *N2* = 0 (zero), then let *CRL* be a zero-length string; otherwise, let *CRL* be:

CR_1 AS CRI_1, CR_2 AS CRI_2, ..., CR_{N2} AS CRI_{N2}

n) Let *N3* be the number of <derived column>s simply contained in *SL* that do not specify <as clause>.

o) Let $DCOL_k$, 1 (one) ≤ k ≤ $N3$, be the <derived column>s simply contained in *SL* that do not specify an <as clause>. For each k, let $COLN_k$ be the <column name> determined as follows.

Case:

i) If $DCOL_k$ is a single column reference, then let $COLN_k$ be the <column name> of the column designated by the column reference.

ii) Otherwise, let $COLN_k$ be an implementation-dependent <column name>,

p) Let *SL2* be obtained from *SL* by replacing each <derived column> $DCOL_k$ by

```
DCOLk AS COLNk
```

q) Let *GWQN* be an arbitrary <identifier>.

r) Let *SLNEW* be the <select list> obtained from *SL2* by replacing each simply contained <set function specification> SFS_i by $GWQN.SFSI_i$ and replacing each <column reference> CR_j that is contained without an intervening <subquery> or <set function specification> by $GWQN.CRI_j$.

s) Let *WICNEW* be the <window clause> obtained from *WIC* by replacing each <set function specification> SFS_i by $GWQN.SFSI_i$ and by replacing each <column reference> CR_j by $GWQN.CRI_j$.

t) If either *SFSL* or *CRL* is a zero-length string, then let *COMMA* be a zero-length string; otherwise, let *COMMA* be "," (a <comma>).

u) *GWQ* is equivalent to the following <query specification>:

```
SELECT SLNEW
FROM ( SELECT SQ SFSL COMMA CRL
       FC
       WC
       GBC
       HC ) AS GWQN
WICNEW
```

13) A <query specification> is *possibly non-deterministic* if any of the following conditions are true:

a) The <set quantifier> DISTINCT is specified and one of the columns of *T* has a data type of character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE.

b) The <query specification> generally contains a <value expression>, <query specification>, or <query expression> that is possibly non-deterministic.

c) The <select list>, <having clause>, or <window clause> contains a reference to a column *C* of *T* that has a data type of character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE, and the functional dependency $G \mapsto C$, where *G* is the set consisting of the grouping columns of *T*, holds in *T*.

14) If <table expression> does not immediately contain a <group by clause> and <table expression> is simply contained in a <query expression> that is the aggregation query of some <set function specification>, then `GROUP BY ()` is implicit.

NOTE 164 — "aggregation query" is defined in Subclause 6.9, "<set function specification>".

15) If *T* is a grouped table, then let *G* be the set of grouping columns of *T*. In each <value expression> contained in <select list> , each column reference that references a column of *T* shall reference some column *C* that is functionally dependent on *G* or shall be contained in an aggregated argument of a <set function specification> whose aggregation query is *QS*.

NOTE 165 — See also the Syntax Rules of Subclause 6.7, "<column reference>".

16) Each column of *TQS* has a column descriptor that includes a data type descriptor that is the same as the data type descriptor of the <value expression> simply contained in the <derived column> defining that column.

17) Case:

a) If the *i*-th <derived column> in the <select list> specifies an <as clause> that contains a <column name> *CN*, then the <column name> of the *i*-th column of the result is *CN*.

b) If the *i*-th <derived column> in the <select list> does not specify an <as clause> and the <value expression> of that <derived column> is a single column reference, then the <column name> of the *i*-th column of the result is the <column name> of the column designated by the column reference.

c) Otherwise, the <column name> of the *i*-th column of the <query specification> is implementation-dependent.

18) A column of *TQS* is *known not null* if and only if at least one of the following conditions applies:

a) It is not defined by a <derived column> containing any of the following:

i) A column reference for a column *C* that is possibly nullable.

ii) An <indicator parameter>.

iii) An <indicator variable>.

iv) A <dynamic parameter specification>.

v) An SQL parameter.

vi) A <routine invocation>, <method reference>, or <method invocation> whose subject routine is an SQL-invoked routine that either is an SQL routine or is an external routine that specifies or implies PARAMETER STYLE SQL.

vii) A <subquery>.

viii) CAST (NULL AS *X*) (where *X* represents a <data type> or a <domain name>).

ix) A <window function> whose <window function type> does not contain <rank function type>, ROW_NUMBER, or an <aggregate function> that simply contains COUNT.

x) CURRENT_USER, CURRENT_ROLE, or SYSTEM_USER.

xi) A <set function specification> that does not contain COUNT.

xii) A <case expression>.

xiii) A <field reference>.

xiv) An <array element reference>.

xv) A <multiset element reference>.

xvi) A <dereference operation>.

xvii) A <reference resolution>.

xviii) A <comparison predicate>, <between predicate>, <in predicate>, or <quantified comparison predicate> *P* such that the declared type of a field of a <row value predicand> that is simply contained in *P* is a row type, a user-defined type, an array type, or a multiset type.

xix) A <member predicate>.

xx) A <submultiset predicate>.

b) An implementation-defined rule by which the SQL-implementation can correctly deduce that the value of the column cannot be null.

19) Let *TREF* be the <table reference>s that are simply contained in the <from clause> of the <table expression>. The *simply underlying tables* of the <query specification> are the <table or query name>s and <derived table>s contained in *TREF* without an intervening <derived table>.

20) The terms *key-preserving* and *one-to-one* are defined as follows:

a) Let *UT* denote some simply underlying table of *QS*, let *UTCOLS* be the set of columns of *UT*, let *QSCOLS* be the set of columns of *QS*, and let *QSCN* be an exposed <table name> or exposed <correlation name> for *UT* whose scope clause is *QS*.

NOTE 166 — "strong candidate key" is defined in Subclause 4.19, "Candidate keys".

b) *QS* is said to be *key-preserving with respect to UT* if there is some strong candidate key *CKUT* of *UT* such that every member of *CKUT* has some counterpart under *QSCN* in *QSCOLS*.

NOTE 167 — "Counterpart" is defined in Subclause 4.18.2, "General rules and definitions". It follows from this condition that every row in *QS* corresponds to exactly one row in *UT*, namely that row in *UT* that has the same combined value in the columns of *CKUT* as the row in *QS*. There may be more than one row in *QS* that corresponds to a single row in *UT*.

c) *QS* is said to be *one-to-one with respect to UT* if and only if *QS* is key-preserving with respect to *UT*, *UT* is updatable, and there is some strong candidate key *CKQS* of *QS* such that every member of *CKQS* is a counterpart under *UT* of some member of *UTCOLS*.

NOTE 168 — It follows from this condition that every row in *UT* corresponds to at most one row in *QS*, namely that row in *QS* that has the same combined value in the columns of *CKQS* as the row in *UT*.

21) A <query specification> is *potentially updatable* if and only if the following conditions hold:

a) DISTINCT is not specified.

b) Of those <derived column>s in the <select list> that are column references, no column reference appears more than once in the <select list>.

c) The <table expression> immediately contained in *QS* does not simply contain an explicit or implicit <group by clause> or a <having clause>.

22) A <query specification> *QS* is *insertable-into* if and only if every simply underlying table of *QS* is insertable-into.

23) If a <query specification> *QS* is potentially updatable, then

Case:

a) If the <from clause> of the <table expression> specifies exactly one <table reference>, then a column of *QS* is said to be an *updatable column* if it has a counterpart in *TR* that is updatable.

NOTE 169 — The notion of updatable columns of table references is defined in Subclause 7.6, "<table reference>".

b) Otherwise, a column of *QS* is said to be an *updatable column* if it has a counterpart in some column of some simply underlying table *UT* of *QS* such that *QS* is one-to-one with respect to *UT*.

24) A <query specification> is *updatable* if it is potentially updatable and it has at least one updatable column.

25) A <query specification> *QS* is *simply updatable* if it is updatable, the <from clause> immediately contained in the <table expression> immediately contained in *QS* contains exactly one <table reference>, and every result column of *QS* is updatable.

26) The row type *RT* of *TQS* is defined by the sequence of (<field name>, <data type>) pairs indicated by the sequence of column descriptors of *TQS* taken in order.

Access Rules

None.

General Rules

1) If *QS* is contained in a <subquery> *SQ*, then certain <set function specification>s and outer references are resolved, such that their values are constant for every row in the result of *QS*, as follows:

Case:

a) If *SQ* is being evaluated for a given group *G*, then, for every <set function specification> *SFS* contained in *QS* such that the aggregation query of *SFS* simply contains the <table expression> of whose result *G* is a group, the value of *SFS* is the result of evaluating *SFS* for *G*.

b) Otherwise, let *R* be the row for which *SQ* is being evaluated. For every <column reference> *CR* contained in *SQ* that is an outer reference whose qualifying scope is simply contained in a <query specification> that contains *SQ*, the value of *CR* is the value of the field in *R* corresponding to the column referenced by *CR*.

NOTE 170 — An expression having been resolved under this rule is not resolved again in the case where it is contained in a <query expression> contained in *SQ*.

NOTE 171 — The circumstances in which a <subquery> is evaluated for a given group, rather than a given row, are defined in the General Rules of this Subclause and the General Rules of Subclause 7.10, "<having clause>".

2) Case:

a) If *T* is not a grouped table, then each <value expression> is applied to each row of *T* yielding a table *TEMP* of *M* rows, where *M* is the cardinality of *T*. The *i*-th column of the table contains the values derived by the evaluation of the *i*-th <value expression>.

b) If *T* is a grouped table, then

Case:

i) If *T* has 0 (zero) groups, then let *TEMP* be an empty table.

ii) If *T* has one or more groups, then each <value expression> is applied to each group of *T* yielding a table *TEMP* of *M* rows, where *M* is the number of groups in *T*. The *i*-th column of *TEMP* contains the values derived by the evaluation of the *i*-th <value expression>. When a <value expression> is applied to a given group of *T*, that group is the argument source of each <set function specification> in the <value expression>.

3) Case:

a) If the <set quantifier> DISTINCT is not specified, then the result of the <query specification> is *TEMP*.

b) If the <set quantifier> DISTINCT is specified, then the result of the <query specification> is the table derived from *TEMP* by the elimination of all redundant duplicate rows. If the most specific type of any column is character string, datetime with time zone, or a user-defined type, then the precise values in those columns are chosen in an implementationdependent fashion.

Conformance Rules

1) Without Feature F801, "Full set function", conforming SQL language shall not contain a <query specification> that contains more than 1 (one) <set quantifier> that contains DISTINCT, excluding any <subquery> of that <query specification>.

2) Without Feature T051, "Row types", conforming SQL language shall not contain an <all fields reference>.

3) Without Feature T301, "Functional dependencies", in conforming SQL language, if *T* is a grouped table, then in each <value expression> contained in the <select list>, each <column reference> that references a column of *T* shall reference a grouping column or be specified in an aggregated argument of a <set function specification>.

4) Without Feature T111, "Updatable joins, unions, and columns", in conforming SQL language, a <query specification> *QS* is not updatable if it is not simply updatable.

NOTE 172 — If a <set quantifier> DISTINCT is specified, then the Conformance Rules of Subclause 9.10, "Grouping operations", also apply.

5) Without Feature T325, "Qualified SQL parameter references", conforming SQL language shall not contain an <asterisked identifier chain> whose referent is an SQL parameter and whose first <identifier> is the <qualified identifier> of a <routine name>.

6) Without Feature T053, "Explicit aliases for all-fields reference", conforming SQL language shall not contain an <all fields column name list>.

7.13 <query expression>

Function

Specify a table.

Format

```
<query expression> ::=
    [ <with clause> ] <query expression body>

<with clause> ::=
    WITH [ RECURSIVE ] <with list>

<with list> ::=
    <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
    <query name> [ <left paren> <with column list> <right paren> ]
    AS <left paren> <query expression> <right paren> [ <search or cycle clause> ]

<with column list> ::= <column name list>

<query expression body> ::=
    <query term>
  | <query expression body> UNION [ ALL | DISTINCT ]
    [ <corresponding spec> ] <query term>
  | <query expression body> EXCEPT [ ALL | DISTINCT ]
    [ <corresponding spec> ] <query term>

<query term> ::=
    <query primary>
  | <query term> INTERSECT [ ALL | DISTINCT ]
    [ <corresponding spec> ] <query primary>

<query primary> ::=
    <simple table>
  | <left paren> <query expression body> <right paren>

<simple table> ::=
    <query specification>
  | <table value constructor>
  | <explicit table>

<explicit table> ::= TABLE <table or query name>

<corresponding spec> ::=
    CORRESPONDING [ BY <left paren> <corresponding column list> <right paren> ]

<corresponding column list> ::= <column name list>
```

Syntax Rules

1) Let *QE* be the <query expression>.

2) If <with clause> is specified, then:

   a) If a <with clause> *WC* immediately contains RECURSIVE, then *WC*, its <with list>, and its <with list element>s are said to be *potentially recursive*. Otherwise they are said to be *non-recursive*.

   b) Let *n* be the number of <with list element>s. For each *i* , 1 (one) ≤ *i* < *n*, for each *j*, *i* < *j* ≤ *n*, the *j*-th <with list element> shall not immediately contain a <query name> that is equivalent to the <query name> immediately contained in the *i*-th <with list element>.

   c) If the <with clause> is non-recursive, then for all *i* between 1 (one) and *n*, the scope of the <query name> *WQN* immediately contained in the *i*-th <with list element> WLE_i is the <query expression> immediately contained in every <with list element> WLE_k, where *k* ranges from *i*+1 to *n*, and the <query expression body> immediately contained in <query expression>. A <table or query name> contained in this scope that immediately contains *WQN* is a *query name in scope*.

   d) If the <with clause> is potentially recursive, then for all *i* between 1 (one) and *n*, the scope of the <query name> *WQN* immediately contained in the *i*-th <with list element> WLE_i is the <query expression> immediately contained in every <with list element> WLE_k, where *k* ranges from 1 (one) to *n*, and the <query expression body> immediately contained in <query expression>. A <table or query name> contained in this scope that immediately contains *WQN* is a *query name in scope*.

   e) For every <with list element> *WLE*, let *WQE* be the <query expression> specified by *WLE* and let *WQT* be the table defined by *WQE*.

      i) If any two columns of *WQT* have equivalent names or if *WLE* is potentially recursive, then *WLE* shall specify a <with column list>. If *WLE* specifies a <with column list> *WCL*, then:

         1) Equivalent <column name>s shall not be specified more than once in *WCL*.

         2) The number of <column name>s in *WCL* shall be the same as the degree of *WQT*.

      ii) Every column of a character string type in *WQT* shall have a declared type collation.

   f) A *query name dependency graph QNDG* of a potentially recursive <with list> *WL* is a directed graph such that, for *i* ranging from 1 (one) to the number of <query name>s simply contained in *WL*:

      i) Each node represents a <query name> WQN_i immediately contained in a <with list element> WLE_i of *WL*.

      ii) Each arc from a node WQN_i to a node WQN_j represents the fact that WQN_j is referenced by a <query name> contained in the <query expression> immediately contained in WLE_i. WQN_i is said to *depend immediately* on WQN_j.

   g) For a potentially recursive <with list> *WL* with *n* elements, and for *i* ranging from 1 (one) to *n*, let WLE_i be the *i*-th <with list element> of *WL*, let WQN_i be the <query name> immediately contained in WLE_i, let WQE_i be the <query expression> immediately contained in WLE_i, let WQT_i be the table defined by WQE_i, and let *QNDG* be the query name dependency graph of *WL*.

i) *WL* is said to be *recursive* if *QNDG* contains at least one cycle.

Case:

1) If *QNDG* contains an arc from WQN_i to itself, then WLE_i, WQN_i, and WQT_i are said to be *recursive*. WQN_i is said to belong to the *stratum* of WQE_i.

2) If *QNDG* contains a cycle comprising WQN_i, ..., WQN_k, with $k \neq i$, then it is said that WQN_i, ..., WQN_k are *recursive* and *mutually recursive* to each other, WQT_i, ..., WQT_k are *recursive* and *mutually recursive* to each other, and WLE_i, ..., WLE_k are *recursive* and *mutually recursive* to each other.

   For each j ranging from i to k, WQN_j belongs to the stratum of WQE_i,..., and WQE_k.

3) Among the WQE_i, ... WQE_k of a given stratum, there shall be at least one <query expression>, say WQE_j, such that:

   A) WQE_j is a <query expression body> that immediately contains UNION.

   B) WQE_j has one operand that does not contain a <query name> referencing any of WQN_i, ..., WQN_k. This operand is said to be the *non-recursive operand* of WQE_j.

   C) WQE_j is said to be an *anchor expression*, and WQN_j an *anchor name*.

   D) Let *CCCG* be the subgraph of *QNDG* that contains no nodes other than WQN_i, ..., WQN_k. For any anchor name WQN_j, remove the arcs to those query names WQN_l that are referenced by any <query name> contained in WQE_j. The remaining graph *SCCGP* shall not contain a cycle.

ii) If WLE_i is recursive, then

Case:

1) If WQE_i contains at most one WQN_k that belongs to the stratum of WQE_i, then WLE_i is *linearly recursive*.

2) Otherwise, let WQE_i contain any two <query name>s referencing WQN_k and WQN_l, both of which belong to the stratum of WQE_i.

   Case:

   A) WLE_i is *linearly recursive* if each of the following conditions is satisfied:

      I) WQE_i does not contain a <table reference list> that contains <query name>s referencing both WQN_k and WQN_l.

      II) WQE_i does not contain a <joined table> such that *TR1* and *TR2* are the first and second <table reference>s, respectively, and *TR1* and *TR2* contain <query name>s referencing WQN_k and WQN_l, respectively.

III) WQE_i does not contain a <table expression> that immediately contains a <from clause> that contains WQN_k, and immediately contains a <where clause> containing a <subquery> that contains a <query name> referencing WQN_l.

B) Otherwise, WLE_i is said to be *non-linearly recursive*.

iii) For each WLE_i, for i ranging from 1 (one) to n, and for each WQN_j that belongs to the stratum of WQE_i:

1) WQE_i shall not contain a <query expression body> that contains a <query name> referencing WQN_j and immediately contains EXCEPT where the right operand of EXCEPT contains WQN_j.

2) WQE_i shall not contain a <routine invocation> with an <SQL argument list> that contains one or more <SQL argument>s that immediately contain a <value expression> that contains a <query name> referencing WQN_j.

3) WQE_i shall not contain a <table subquery> *TSQ* that contains a <query name> referencing WQN_j, unless *TSQ* is a <derived table> that is immediately contained in a <table primary> that is immediately contained in a <table reference> that is immediately contained in a <from clause> that is immediately contained in a <table expression> that is immediately contained in a <query specification> that constitutes a <simple table> that constitutes a <query primary> that constitutes a <query term> that is immediately contained in a <query expression body> that is WQE_i.

4) WQE_i shall not contain a <query specification> *QS* such that:

A) *QS* immediately contains a <table expression> *TE* that contains a <query name> referencing WQN_j, and

B) *QS* immediately contains a <select list> *SL* or *TE* immediately contains a <having clause> *HC* and *SL* or *TE* contain a <set function specification>.

5) WQE_i shall not contain a <query expression body> that contains a <query name> referencing WQN_j and simply contains INTERSECT ALL or EXCEPT ALL.

6) WQE_i shall not contain a <qualified join> *QJ* in which:

A) *QJ* immediately contains a <join type> that specifies FULL and a <table reference> or <table factor> that contains a <query name> referencing WQN_j.

B) *QJ* immediately contains a <join type> that specifies LEFT and a <table factor> following the <join type> that contains a <query name> referencing WQN_j.

C) *QJ* immediately contains a <join type> that specifies RIGHT and a <table reference> preceding the <join type> that contains a <query name> referencing WQN_j.

7) WQE_i shall not contain a <natural join> *QJ* in which:

A) *QJ* immediately contains a <join type> that specifies FULL and a <table reference> or <table primary> that contains a <query name> referencing WQN_j.

B) *QJ* immediately contains a <join type> that specifies LEFT and a <table primary> following the <join type> that contains a <query name> referencing WQN_j.

C) *QJ* immediately contains a <join type> that specifies RIGHT and a <table reference> preceding the <join type> that contains a <query name> referencing WQN_j.

iv) If WLE_i is recursive, then WLE_i shall be linearly recursive.

v) WLE_i is said to be *expandable* if all of the following are true:

1) WLE_i is recursive.

2) WLE_i is linearly recursive.

3) WQE_i is a <query expression body> that immediately contains UNION or UNION ALL. Let $WQEB_i$ be the <query expression body> immediately contained in WQE_i. Let QEL_i and QTR_i be the <query expression body> and the <query term> immediately contained in $WQEB_i$. WQN_i shall not be contained in QEL_i, and QTR_i shall be a <query specification>.

4) WQN_i is not mutually recursive.

h) If a <with list element> *WLE* is not expandable, then it shall not immediately contain a <search or cycle clause>.

3) Let *T* be the table specified by the <query expression>.

4) The <explicit table>

```
TABLE <table or query name>
```

is equivalent to the <query expression>

```
( SELECT * FROM <table or query name> )
```

5) Let *set operator* be UNION ALL, UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT.

6) If UNION, EXCEPT, or INTERSECT is specified and neither ALL nor DISTINCT is specified, then DISTINCT is implicit.

7) <query expression> *QE1* is *updatable* if and only if for every <query expression> or <query specification> *QE2* that is simply contained in *QE1*:

a) *QE1* contains *QE2* without an intervening <query expression body> that specifies UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.

b) If *QE1* simply contains a <query expression body> *QEB* that specifies UNION ALL, then:

i) *QEB* immediately contains a <query expression> *LO* and a <query term> *RO* such that no leaf generally underlying table of *LO* is also a leaf generally underlying table of *RO*.

ii) For every column of *QEB*, the underlying columns in the tables identified by *LO* and *RO*, respectively, are either both updatable or not updatable.

c) *QE1* contains *QE2* without an intervening <query term> that specifies INTERSECT.

d) *QE2* is updatable.

8) A table specified by a <query name> immediately contained in a <with list element> *WLE* is *updatable* if and only if the <query expression> simply contained in *WLE* is updatable.

9) <query expression> *QE1* is *insertable-into* if and only if *QE1* simply contains exactly one <query expression> or <query specification> *QE2* and *QE2* is insertable-into.

10) A table specified by a <query name> immediately contained in a <with list element> *WLE* is *insertable-into* if and only if the <query expression> simply contained in *WLE* is insertable-into.

11) For every <simple table> *ST* contained in *QE*,

Case:

a) If *ST* is a <query specification> *QS*, then the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of *QS*.

b) If *ST* is an <explicit table> *ET*, then the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of the table identified by the <table or query name> contained in *ET*.

c) Otherwise, the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of the <table value constructor> immediately contained in *ST*.

12) For every <query primary> *QP* contained in *QE*,

Case:

a) If *QP* is a <simple table> *ST*, then the column descriptor of each column of *QP* is the same as the column descriptor of the corresponding column of *ST*.

b) Otherwise, the column descriptor of each column of *QP* is the same as the column descriptor of the corresponding column of the <query expression body> immediately contained in *QP*.

13) If a set operator is specified in a <query term> or a <query expression body>, then:

a) Let *T1*, *T2*, and *TR* be respectively the first operand, the second operand, and the result of the <query term> or <query expression body>.

b) Let *TN1* and *TN2* be the effective names for *T1* and *T2*, respectively.

c) If the set operator is UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT, then each column of *T1* and *T2* is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

14) If a set operator is specified in a <query term> or a <query expression body>, then let *OP* be the set operator.

Case:

a) If CORRESPONDING is specified, then:

i) Within the columns of *T1*, equivalent <column name>s shall not be specified more than once and within the columns of *T2*, equivalent <column name>s shall not be specified more than once.

ii) At least one column of *T1* shall have a <column name> that is the <column name> of some column of *T2*.

iii) Case:

1) If <corresponding column list> is not specified, then let *SL* be a <select list> of those <column name>s that are <column name>s of both *T1* and *T2* in the order that those <column name>s appear in *T1*.

2) If <corresponding column list> is specified, then let *SL* be a <select list> of those <column name>s explicitly appearing in the <corresponding column list> in the order that these <column name>s appear in the <corresponding column list>. Every <column name> in the <corresponding column list> shall be a <column name> of both *T1* and *T2*.

iv) The <query term> or <query expression body> is equivalent to:

```
( SELECT SL FROM TN1 ) OP ( SELECT SL FROM TN2 )
```

b) If CORRESPONDING is not specified, then *T1* and *T2* shall be of the same degree.

15) If a <query term> is a <query primary>, then the declared type of the <query term> is that of the <query primary>. The column descriptor of the *i*-th column of the <query term> is the same as the column descriptor of the *i*-th column of the <query primary>.

16) If a <query term> immediately contains a set operator, then:

a) Let *C* be the <column name> of the *i*-th column of *T1*. If the <column name> of the *i*-th column of *T2* is *C*, then the <column name> of the *i*-th column of *TR* is *C*; otherwise, the <column name> of the *i*-th column of *TR* is implementation-dependent.

b) The declared type of the *i*-th column of *TR* is determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of the *i*-th column of *T1* and the *i*-th column of *T2*. If the *i*-th columns of either *T1* or *T2* are known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

17) If a <query term> is a <query primary>, then the column descriptors of the <query term> are the same as the column descriptors of the <query primary>.

18) Case:

a) If a <query expression body> is a <query term>, then the column descriptors of the <query expression body> are the same as the column descriptors of the <query term>.

b) If a <query expression body> immediately contains a set operator, then:

i) Let *C* be the <column name> of the *i*-th column of *T1*. If the <column name> of the *i*-th column of *T2* is *C*, then the <column name> of the *i*-th column of *TR* is *C*; otherwise, the <column name> of the *i*-th column of *TR* is implementation-dependent.

ii) If *TR* is not the result of an anchor expression, then the declared type of the *i*-th column of *TR* is determined by applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the *i*-th column of *T1* and the *i*-th column of *T2*.

Case:

1) If the <query expression body> immediately contains EXCEPT, then if the *i*-th column of *T1* is known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

2) Otherwise, if the *i*-th columns of both *T1* and *T2* are known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

iii) If *TR* is the result of an anchor expression *ARE*, then:

1) Let *l* be the number of recursive tables that belong to the stratum of *ARE*. For *j* ranging from 1 (one) to *l*, let WQT_j be those tables. Of the operands *T1* and *T2* of *TR*, let *TNREC* be the operand that is the result of the non-recursive operand of *ARE* and let *TREC* be the other operand. The *i*-th column of *TR* is said to be *recursively referred to* if there exists at least one *k*, $1 \le k \le l$, such that a column of WQT_k is an underlying column of the *i*-th column of *TREC*. Otherwise, that column is said to be *not recursively referred to*.

2) If the *i*-th column of *TR* is not recursively referred to, then the declared type of the *i*-th column of *TR* is determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of the *i*-th column of *T1* and the *i*-th column of *T2*. If the *i*-th columns of either *T1* or *T2* are known not nullable, then the *i*-th column of *TR* is *known not nullable*; otherwise, the *i*-th column of *TR* is *possibly nullable*.

3) If the *i*-th column of *TR* is recursively referred to, then:

A) The *i*-th column of *TR* is *possibly nullable*.

B) Case:

I) If *T1* is *TNREC*, then if the *i*-th column of *TR* is recursively referred to, then the declared type of the *i*-th column of *TR* is the same as the declared type of the *i*-th column of *T1*.

II) If *T2* is *TNREC*, then if the *i*-th column of *TR* is recursively referred to, then the declared type of the *i*-th column of *TR* is the same as the declared type of the *i*-th column of *T2*.

19) The *simply underlying table*s of *QE* are the <table or query name>s, <query specification>s, and <derived table>s contained, without an intervening <derived table> or an intervening <join condition>, in the <query expression body> immediately contained in *QE*.

20) An <explicit table> is *possibly non-deterministic* if the simply contained <table or query name> identifies a viewed table whose original <query expression> is possibly non-deterministic.

21) A <query expression> is *possibly non-deterministic* if any of the following are true:

a) The <query expression> is a <query primary> that is possibly non-deterministic.

b) UNION, EXCEPT, or INTERSECT is specified and either of the first or second operands is possibly non-deterministic.

c) UNION, EXCEPT, or INTERSECT is specified and there is a column of the result such that the declared types *DT1* and *DT2* of the column in the two operands have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

d) Both of the following are true:

i) *T* contains a set operator UNION and ALL is not specified, or *T* contains either of the set operators EXCEPT or INTERSECT.

ii) Exactly one of the following is true:

1) The first or second operand contains a column that has a declared type of character string.

2) The first or second operand contains a column that has a declared type of datetime with time zone.

3) The first or second operand contains a column that has a declared type that is a user-defined type.

22) The *underlying columns* of each column of *QE* and of *QE* itself are defined as follows:

a) A column of a <table value constructor> has no underlying columns.

b) The underlying columns of every *i*-th column of a <simple table> *ST* are the underlying columns of the *i*-th column of the table immediately contained in *ST*. A column of *ST* is called an *updatable column* of *ST* if the underlying column of *ST* is updatable; otherwise, this column is *not updatable*.

c) If no set operator is specified, then the underlying columns of every *i*-th column of *QE* are the underlying columns of the *i*-th column of the <simple table> simply contained in *QE*. A column of such a *QE* is called an *updatable column* of *QE* if its underlying column is updatable; otherwise, this column is *not updatable*.

d) If a set operator is specified, then the underlying columns of every *i*-th column of *QE* are the underlying columns of the *i*-th column of *T1* and those of the *i*-th column of *T2*.

Case:

i) If a set operator UNION ALL is specified and both underlying columns of *T1* and *T2* of the *i*-th column of *QE* are updatable, then the *i*-th column of *QE* is an *updatable column* of *QE*.

ii) Otherwise, the *i*-th column of *QE* is not updatable.

NOTE 173 — If a set operator UNION DISTINCT, EXCEPT, or INTERSECT is specified, then there are no updatable columns.

e) Let *C* be some column. *C* is an underlying column of *QE* if and only if *C* is an underlying column of some column of *QE*.

23) A <query expression> *QE* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

Access Rules

None.

General Rules

1) If a non-recursive <with clause> is specified, then:

a) For every <with list element> *WLE*, let *WQN* be the <query name> immediately contained in *WLE*. Let *WQE* be the <query expression> immediately contained in *WLE*. Let *WLT* be the table resulting from evaluation of *WQE*, with each column name replaced by the corresponding element of the <with column list>, if any, immediately contained in *WLE*.

b) Every <table reference> contained in <query expression> that specifies *WQN* identifies *WLT*.

2) If a potentially recursive <with clause> *WC* is specified, then:

a) Let *n* be the number of <with list element>s WLE_i of the <with list> *WL* immediately contained in *WC*. For *i* ranging from 1 (one) to *n*, let WQN_i and WQE_i be the <query name>s and the <query expression>s immediately contained in WLE_i. Let WLP_j be the elements of a partitioning of *WL* such that each WLP_j contains all WLE_i that belong to one stratum, and let *m* be the number of partitions. Let the *partition dependency graph PDG* of *WL* be a directed graph such that:

i) Each partition WLP_j of *WL* is represented by exactly one node of *PDG*.

ii) There is an arc from the node representing WLP_j to the node representing WLP_k if and only if WLP_j contains at least one WLE_i, WLP_k contains at least one WLE_h, and WQE_i contains a <query name> referencing WQN_h.

b) While the set of nodes of *PDG* is not empty, do:

i) Evaluate the partitions of *PDG* that have no outgoing arc.

ii) Remove the partitions and their incoming arcs from *PDG*.

c) Let *LIP* be some partition of *WL*. Let *m* be the number of <with list element>s in *LIP*, and for *i* ranging from 1 (one) to *m*, let WLE_i be a <with list element> of *LIP*, and let WQN_i and WQE_i be the <query name> and <query expression> immediately contained in WLE_i. Let SQE_i be the set of <query expression>s contained in WQE_i. Let *SQE* be a set of <query expression>s such that a <query expression> belongs to *SQE* if and only if it is contained in some WQE_i. Let *p* be the number of <query expression>s in *SQE* and let AQE_i, $1 \le k \le p$ be the *k*-th <query expression> belonging to *SQE*.

i) Every <query expression> AQE_k that contains a recursive query name in scope is marked as *recursive*.

ii) Let RT_k and WT_k be tables whose row type is the row type of AQE_k. Let RT_k and WT_k be initially empty. RT_k and WT_k are said to be *associated with* AQE_k. If AQE_k is immediately contained in some WQE_i, then RT_k and WT_k are said to be the *intermediate result table* and *working table*, respectively, *associated with* the <query name> WQN_i.

iii) If a <query expression> AQE_k not marked as recursive is immediately contained in a <query expression body> that is marked as recursive and that specifies UNION, then AQE_i is marked as *iteration ignorable*.

iv) For each AQE_k,

Case:

1) If AQE_k consists of a <query specification> that immediately contains DISTINCT, then AQE_k *suppresses duplicates*.

2) If AQE_k consists of a <query expression body> or <query term> that explicitly or implicitly immediately contains DISTINCT, then AQE_k *suppresses duplicates*.

3) Otherwise, AQE_k does not suppress duplicates.

v) If an AQE_k is not marked as recursive, then let RT_k and WT_k be the result of AQE_k.

vi) For every RT_k, let RTN_k be the name of RT_k. If AQE_k is not marked as recursive, then replace AQE_k with:

```
TABLE RTN_k
```

vii) For every WQE_i of *LIP*, let the *recursive query names in scope* denote the associated result tables. Evaluate every WQE_i. For every AQE_k contained in any such WQE_i, let RT_k and WT_k be the result of AQE_k.

NOTE 174 — This ends the initialization phase of the evaluation of a partition.

viii) For every AQE_k of *LIP* that is marked as iteration ignorable, let RT_k be an empty table.

ix) While some WT_k of *LIP* is not empty, do:

1) Let the recursive query names in scope of *LIP* denote the associated working tables.

2) Evaluate every WQE_i of *LIP*.

3) For every AQE_k that is marked as recursive,

Case:

A) If AQE_k suppresses duplicates, then let WT_k be the result of AQE_k EXCEPT RTN_k.

B) Otherwise, let WT_k be the result of AQE_k.

4) For every WT_k, let WTN_k be the table name of WT_k. Let RT_k be the result of:

```
TABLE WTN_k UNION ALL TABLE RTN_k
```

x) Any reference to WQN_i identifies the intermediate result table RT_k associated with WQN_i.

3) If a set operator is specified, then for each column *C* of *T*, let *UDT* be the declared type of *C* and let *SV* be the value of the column corresponding to *C* in each row of each operand. The value of *C* in the corresponding row of *T* is

```
CAST ( SV AS UDT )
```

4) Case:

a) If no set operator is specified, then *T* is the result of the specified <simple table> or <joined table>.

b) If a set operator is specified, then the result of applying the set operator is a table containing the following rows:

i) Let *R* be a row that is a duplicate of some row in *T1* or of some row in *T2* or both. Let *m* be the number of duplicates of *R* in *T1* and let *n* be the number of duplicates of *R* in *T2*, where $m \geq 0$ and $n \geq 0$.

ii) If DISTINCT is specified or implicit, then

Case:

1) If UNION is specified, then

Case:

A) If $m > 0$ or $n > 0$, then *T* contains exactly one duplicate of *R*.

B) Otherwise, *T* contains no duplicate of *R*.

2) If EXCEPT is specified, then

Case:

A) If $m > 0$ and $n = 0$, then *T* contains exactly one duplicate of *R*.

B) Otherwise, *T* contains no duplicate of *R*.

3) If INTERSECT is specified, then

Case:

A) If $m > 0$ and $n > 0$, then *T* contains exactly one duplicate of *R*.

B) Otherwise, *T* contains no duplicates of *R*.

iii) If ALL is specified, then

Case:

1) If UNION is specified, then the number of duplicates of *R* that *T* contains is $(m + n)$.

2) If EXCEPT is specified, then the number of duplicates of *R* that *T* contains is the maximum of $(m - n)$ and 0 (zero).

3) If INTERSECT is specified, then the number of duplicates of *R* that *T* contains is the minimum of *m* and *n*.

NOTE 175 — See the General Rules of Subclause 8.2, "<comparison predicate>".

5) Case:

a) If EXCEPT is specified and a row *R* of *T* is replaced by some row *RR*, then the row of *T1* from which *R* is derived is replaced by *RR*.

b) If INTERSECT is specified, then:

i) If a row *R* is inserted into *T*, then:

1) If *T1* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T1*.

2) If *T1* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T1*.

3) If *T2* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T2*.

4) If *T2* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T2*.

ii) If a row *R* is replaced by some row *RR*, then:

1) The row of *T1* from which *R* is derived is replaced with *RR*.

2) The row of *T2* from which *R* is derived is replaced with *RR*.

Conformance Rules

1) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", in conforming SQL language, a <query expression> shall not contain a <with clause>.

2) Without Feature T122, "WITH (excluding RECURSIVE) in subquery", in conforming SQL language, a <query expression> contained in a <subquery>, a <multiset value constructor by query>, or an <array value constructor by query> shall not contain a <with clause>.

3) Without Feature T131, "Recursive query", conforming SQL language shall not contain a <query expression> that contains RECURSIVE.

4) Without Feature T132, "Recursive query in subquery", in conforming SQL language, a <query expression> contained in a <subquery>, a <multiset value constructor by query>, or an <array value constructor by query> shall not contain RECURSIVE.

5) Without Feature F661, "Simple tables", conforming SQL language shall not contain a <simple table> that immediately contains a <table value constructor> except in an <insert statement>.

6) Without Feature F661, "Simple tables", conforming SQL language shall not contain an <explicit table>.

7) Without Feature F302, "INTERSECT table operator", conforming SQL language shall not contain a <query term> that contains INTERSECT.

8) Without Feature F301, "CORRESPONDING in query expressions", conforming SQL language shall not contain a <query expression> that contains CORRESPONDING.

9) Without Feature T551, "Optional key words for default syntax", conforming SQL language shall not contain UNION DISTINCT, EXCEPT DISTINCT, or INTERSECT DISTINCT.

10) Without Feature T111, "Updatable joins, unions, and columns", in conforming SQL language, a <query expression body> that immediately contains UNION is not updatable.

11) Without Feature F304, "EXCEPT ALL table operator", conforming SQL language shall not contain a <query expression> that contains EXCEPT ALL.

NOTE 176 — If DISTINCT, INTERSECT or EXCEPT is specified, then the Conformance Rules of Subclause 9.10, "Grouping operations", apply.

7.14 <search or cycle clause>

Function

Specify the generation of ordering and cycle detection information in the result of recursive query expressions.

Format

```
<search or cycle clause> ::=
    <search clause>
  | <cycle clause>
  | <search clause> <cycle clause>

<search clause> ::=
    SEARCH <recursive search order> SET <sequence column>

<recursive search order> ::=
    DEPTH FIRST BY <sort specification list>
  | BREADTH FIRST BY <sort specification list>

<sequence column> ::= <column name>

<cycle clause> ::=
    CYCLE <cycle column list> SET <cycle mark column> TO <cycle mark value>
    DEFAULT <non-cycle mark value> USING <path column>

<cycle column list> ::=
    <cycle column> [ { <comma> <cycle column> }... ]

<cycle column> ::= <column name>

<cycle mark column> ::= <column name>

<path column> ::= <column name>

<cycle mark value> ::= <value expression>

<non-cycle mark value> ::= <value expression>
```

Syntax Rules

1) Let *WLEC* be an expandable <with list element> immediately containing a <search or cycle clause>.

2) Let *WQN* be the <query name>, *WCL* the <with column list>, and *WQE* the <query expression> immediately contained in *WLEC*. Let *WQEB* be the <query expression body> immediately contained in *WQE*. Let *OP* be the set operator immediately contained in *WQEB*. Let *TLO* be the <query expression body> that constitutes the first operand of *OP* and let *TRO* be the <query specification> that (necessarily) constitutes the second operand of *OP*.

   a) Let *TROSL* be the <select list> immediately contained in *TRO*. Let *WQNTR* be the <table reference> simply contained in the <from clause> immediately contained in the <table expression> *TROTE* immediately contained in *TRO* such that *WQNTR* immediately contains *WQN*.

Case:

i) If *WQNTR* simply contains a <correlation name>, then let *WQNCRN* be that <correlation name>.

ii) Otherwise, let *WQNCRN* be *WQN*.

b) Case:

i) If *WLEC* simply contains a <search clause> *SC*, then let *SQC* be the <sequence column> and *SO* be the <recursive search order> immediately contained in *SC*. Let *SPL* be the <sort specification list> immediately contained in *SO*.

1) *WCL* shall not contain a <column name> that is equivalent to *SQC*.

2) Every <column name> of *SPL* shall be equivalent to some <column name> contained in *WCL*. No <column name> shall be contained more than once in *SPL*.

3) Case:

A) If *SO* immediately contains DEPTH, then let *SCEX1* be:

```
WQNCRN.SQC
```

let *SCEX2* be:

```
SQC || ARRAY [ROW(SPL)]
```

and let *SCIN* be:

```
ARRAY [ROW(SPL)]
```

B) If *SO* immediately contains BREADTH, then let *SCEX1* be:

```
( SELECT OC.*
FROM ( VALUES (WQNCRN.SQC) )
OC(LEVEL, SPL) )
```

let SCEX2 be:

```
ROW(SQC.LEVEL + 1, SPL)
```

and let *SCIN* be:

```
ROW(0, SPL)
```

ii) If *WLEC* simply contains a <cycle clause> *CC*, then let *CCL* be the <cycle column list>, let *CMC* be the <cycle mark column>, let *CMV* be the <cycle mark value>, let *CMD* be the <non-cycle mark value>, and let *CPA* be the <path column> immediately contained in *CC*.

1) Every <column name> of *CCL* shall be equivalent to some <column name> contained in *WCL*. No <column name> shall be contained more than once in *CCL*.

2) *CMC* and *CPA* shall not be equivalent to each other and not equivalent to any <column name> of *WCL*.

3) The declared type of *CMV* and *CMD* shall be character string of length 1 (one). *CMV* and *CMD* shall be literals and *CMV* shall not be equal to *CMD*.

4) Let *CCEX1* be:

```
WQNCRN.CMC, WQNCRN.CPA
```

Let *CCEX2* be:

```
CASE WHEN ROW(CCL) IN (SELECT P.* FROM TABLE(CPA) P)
THEN CMV ELSE CMD END,
CPA || ARRAY [ROW(CCL)]
```

Let *CCIN* be:

```
CMD, ARRAY [ROW(CCL)]
```

Let *NCCON1* be:

```
CMC <> CMV
```

iii) Case:

1) If *WLEC* simply contains a <search clause> and does not simply contain a <cycle clause>, then let *EWCL* be:

```
WCL, SQC
```

Let *ETLOSL* be:

```
WCL, SCIN
```

Let *ETROSL* be:

```
WCL, SCEX2
```

Let *ETROSL1* be:

```
TROSL, SCEX1
```

Let *NCCON* be:

```
TRUE
```

2) If *WLEC* simply contains a <cycle clause> and does not simply contain a <search clause>, then let *EWCL* be:

```
WCL, CMC, CPA
```

Let *ETLOSL* be:

```
WCL, CCIN
```

Let *ETROSL* be:

```
WCL, CCEX2
```

Let *ETROSL1* be:

TROSL, *CCEX1*

Let *NCCON* be:

NCCON1

3) If *WLEC* simply contains both a <search clause> and a <cycle clause> *CC*, then:

A) The <column name>s *SQC*, *CMC*, and *CPA* shall not be equivalent to each other.

B) Let *EWCL* be:

WCL, *SQC*, *CMC*, *CPA*

Let *ETLOSL* be:

WCL, *SCIN*, *CCIN*

Let *ETROSL* be:

WCL, *SCEX2*, *CCEX2*

Let *ETROSL1* be:

TROSL, *SCEX1*, *CCEX1*

C) Let *NCCON* be:

NCCON1

c) *WLEC* is equivalent to the expanded <with list element>:

```
WQN(EWCL) AS
  ( SELECT ETLOSL FROM (TLO) TLOCRN(WCL)
    OP
    SELECT ETROSL
    FROM (SELECT ETROSL1 TROTE) TROCRN(EWCL)
    WHERE NCCON
  )
```

Access Rules

None.

General Rules

None.

Conformance Rules

None.

7.15 <subquery>

Function

Specify a scalar value, a row, or a table derived from a <query expression>.

Format

```
<scalar subquery> ::= <subquery>

<row subquery> ::= <subquery>

<table subquery> ::= <subquery>

<subquery> ::= <left paren> <query expression> <right paren>
```

Syntax Rules

1) The degree of a <scalar subquery> shall be 1 (one).

2) The degree of a <row subquery> shall be greater than 1 (one).

3) Let *QE* be the <query expression> simply contained in <subquery>.

4) The declared type of a <scalar subquery> is the declared type of the column of *QE*.

5) The declared type of a <row subquery> is a row type consisting of one field for each column of *QE*. The declared type and field name of each field of this row type is the declared type and column name of the corresponding column of *QE*.

6) The declared types of the columns of a <table subquery> are the declared types of the respective columns of *QE*.

Access Rules

None.

General Rules

1) Let *OLDSEC* be the most recent statement execution context. A new statement execution context *NEWSEC* is established. *NEWSEC* becomes the most recent statement execution context and is atomic.

2) Let *RS* be a <row subquery>. Let *RRS* be the result of the <query expression> simply contained in *RS*. Let *D* be the degree of *RRS*.

   Case:

   a) If the cardinality of *RRS* is greater than 1 (one), then an exception condition is raised: *cardinality violation*.

b) If the cardinality of *RRS* is 0 (zero), then the value of the <row subquery> is a row whose degree is *D* and whose fields are all the null value.

c) Otherwise, the value of *RS* is *RRS*.

3) Let *SS* be a <scalar subquery>.

Case:

a) If the cardinality of *SS* is greater than 1 (one), then an exception condition is raised: *cardinality violation*.

b) If the cardinality of *SS* is 0 (zero), then the value of the <scalar subquery> is the null value.

c) Otherwise, let *C* be the column of <query expression> simply contained in *SS*. The value of *SS* is the value of *C* in the unique row of the result of the <scalar subquery>.

4) All savepoints that were established during the existence of *NEWSEC* are destroyed. *NEWSEC* ceases to exist and *OLDSEC* becomes the most recent statement execution context.

5) A <subquery> *SQ* that simply contains a <sample clause> returns a table with identical rows for a given set of values for outer references every time *SQ* is evaluated.

NOTE 177 — "Outer reference" is defined in Subclause 6.7, "<column reference>".

Conformance Rules

None.

This page intentionally left blank.

8 Predicates

8.1 <predicate>

Function

Specify a condition that can be evaluated to give a boolean value.

Format

```
<predicate> ::=
    <comparison predicate>
  | <between predicate>
  | <in predicate>
  | <like predicate>
  | <similar predicate>
  | <null predicate>
  | <quantified comparison predicate>
  | <exists predicate>
  | <unique predicate>
  | <normalized predicate>
  | <match predicate>
  | <overlaps predicate>
  | <distinct predicate>
  | <member predicate>
  | <submultiset predicate>
  | <set predicate>
  | <type predicate>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) The result of a <predicate> is the truth value of the immediately contained <comparison predicate>, <between predicate>, <in predicate>, <like predicate>, <similar predicate>, <null predicate>, <quantified comparison predicate>, <exists predicate>, <unique predicate>, <match predicate>, <overlaps predicate>, <distinct predicate>, <member predicate>, <submultiset predicate>, <set predicate>, or <type predicate>.

Conformance Rules

None.

8.2 <comparison predicate>

Function

Specify a comparison of two row values.

Format

```
<comparison predicate> ::= <row value predicand> <comparison predicate part 2>

<comparison predicate part 2> ::= <comp op> <row value predicand>

<comp op> ::=
    <equals operator>
  | <not equals operator>
  | <less than operator>
  | <greater than operator>
  | <less than or equals operator>
  | <greater than or equals operator>
```

Syntax Rules

1) The two <row value predicand>s shall be of the same degree.

2) Let *corresponding fields* be fields with the same ordinal position in the two <row value predicand>s.

3) The declared types of the corresponding fields of the two <row value predicand>s shall be comparable.

4) Let R_x and R_y respectively denote the first and second <row value predicand>s.

5) Let N be the number of fields in the declared type of R_x. Let X_i, 1 (one) $\leq i \leq N$, be the i-th field in the declared type of R_x and let Y_i be the i-th field in the declared type of R_y. For each i:

   a) Case:

      i) If <comp op> is <equals operator> or <not equals operator>, then X_i and Y_i are operands of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

      ii) Otherwise, X_i and Y_i are operands of an ordering operation. The Syntax Rules of Subclause 9.12, "Ordering operations", apply.

   b) Case:

      i) If the declared types of X_i and Y_i are user-defined types, then let *UDT1* and *UDT2* be respectively the declared types of X_i and Y_i. *UDT1* and *UDT2* shall be in the same subtype family. *UDT1* and *UDT2* shall have comparison types.

      NOTE 178 — "Comparison type" is defined in Subclause 4.7.6, "User-defined type comparison and assignment".

      NOTE 179 — The comparison form and comparison categories included in the user-defined type descriptors of both *UDT1* and *UDT2* are constrained to be the same and to be the same as those of all their supertypes. If the comparison

category is either STATE or RELATIVE, then *UDT1* and *UDT2* are constrained to have the same comparison function; if the comparison category is MAP, they are not constrained to have the same comparison function.

ii) If the declared types of X_i and Y_i are reference types, then the referenced type of the declared type of X_i and the referenced type of the declared type of Y_i shall have a common supertype.

iii) If the declared types of X_i and Y_i are collection types in which the declared type of the elements are ET_x and ET_y, respectively, then let *RV1* and *RV2* be <value expression>s whose declared types are respectively ET_x and ET_y. The Syntax Rules of this Subclause are applied to:

RV1 <comp op> *RV2*

iv) If the declared types of X_i and Y_i are row types, then let *RV1* and *RV2* be <value expression>s whose declared types are respectively that of X_i and Y_i. The Syntax Rules of this Subclause are applied to:

RV1 <comp op> *RV2*

6) Let *CP* be the <comparison predicate> "R_x <comp op> R_y".

Case:

a) If the <comp op> is <not equals operator>, then *CP* is equivalent to:

NOT (R_x = R_y)

b) If the <comp op> is <greater than operator>, then *CP* is equivalent to:

(R_y < R_x)

c) If the <comp op> is <less than or equals operator>, then *CP* is equivalent to:

(R_x < R_y
OR
R_y = R_x)

d) If the <comp op> is <greater than or equals operator>, then *CP* is equivalent to:

(R_y < R_x
OR
R_y = R_x)

Access Rules

None.

General Rules

1) Let *XV* and *YV* be two values represented by <value expression>s *X* and *Y*, respectively. The result of:

```
X <comp op> Y
```

is determined as follows:

Case:

a) If either *XV* or *YV* is the null value, then

```
X <comp op> Y
```

is *Unknown*.

b) Otherwise,

Case:

i) If the declared types of *XV* and *YV* are row types with degree *N*, then let X_i, 1 (one) $\leq i \leq N$, denote a <value expression> whose value and declared type is that of the *i*-th field of *XV* and let Y_i denote a <value expression> whose value and declared type is that of the *i*-th field of *YV*. The result of

```
X <comp op> Y
```

is determined as follows:

1) $X = Y$ is *True* if and only if $X_i = Y_i$ is *True* for all *i*.

2) $X < Y$ is *True* if and only if $X_i = Y_i$ is *True* for all $i < n$ and $X_n < Y_n$ for some *n*.

3) $X = Y$ is *False* if and only if NOT ($X_i = Y_i$) is *True* for some *i*.

4) $X < Y$ is *False* if and only if $X = Y$ is *True* or $Y < X$ is *True*.

5) *X* <comp op> *Y* is *Unknown* if *X* <comp op> *Y* is neither *True* nor *False*.

ii) If the declared types of *XV* and *YV* are array types with cardinalities *N1* and *N2*, respectively, then let X_i, 1 (one) $\leq i \leq N1$, denote a <value expression> whose value and declared type is that of the *i*-th element of *XV* and let Y_i denote a <value expression> whose value and declared type is that of the *i*-th element of *YV*. The result of

```
X <comp op> Y
```

is determined as follows:

1) $X = Y$ is *True* if $N1 = 0$ (zero) and $N2 = 0$ (zero).

2) $X = Y$ is *True* if $N1 = N2$ and, for all *i*, $X_i = Y_i$ is *True*.

3) $X = Y$ is *False* if and only if $N1 \neq N2$ or NOT ($X_i = Y_i$) is *True*, for some *i*.

4) *X* <comp op> *Y* is *Unknown* if *X* <comp op> *Y* is neither *True* nor *False*.

iii) If the declared types of *XV* and *YV* are multiset types with cardinalities *N1* and *N2*, respectively, then the result of

```
X <comp op> Y
```

is determined as follows:

Case:

1) *X* = *Y* is *True* if *N1* = *N2*, and there exist an enumeration XVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *XV* and an enumeration YVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *YV* such that for all *j*, $XVE_j = YVE_j$.

2) *X* = *Y* is *Unknown* if *N1* = *N2*, and there exist an enumeration XVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *XV* and an enumeration YVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *YV* such that for all *j*, "$XVE_j = YVE_j$" is either *True* or *Unknown*.

3) Otherwise, *X* = *Y* is *False*.

iv) If the declared types of *XV* and *YV* are user-defined types, then let UDT_x and UDT_y be respectively the declared types of *XV* and *YV*. The result of

```
X <comp op> Y
```

is determined as follows:

1) If the comparison category of UDT_x is MAP, then let *HF1* be the <routine name> with explicit <schema name> of the comparison function of UDT_x and let *HF2* be the <routine name> with explicit <schema name> of the comparison function of UDT_y. If *HF1* identifies an SQL-invoked method, then let *HFX* be *X.HF1*; otherwise, let *HFX* be *HF1*(*X*). If *HF2* identifies an SQL-invoked method, then let *HFY* be *Y.HF2*; otherwise, let *HFY* be *HF2*(*Y*).

```
X <comp op> Y
```

has the same result as

```
HFX <comp op> HFY
```

2) If the comparison category of UDT_x is RELATIVE, then:

A) Let *RF* be the <routine name> with explicit <schema name> of the comparison function of UDT_x.

B) `X = Y`

has the same result as

```
RF (X, Y ) = 0
```

C) `X < Y`

has the same result as

```
RF (X, Y ) = -1
```

D) `X <> Y`

has the same result as

```
RF (X, Y ) <> 0
```

E) `X > Y`

has the same result as

```
RF (X, Y ) = 1
```

F) `X <= Y`

has the same result as

```
RF (X, Y ) = -1 OR RF (X, Y ) = 0
```

G) `X >= Y`

has the same result as

```
RF (X, Y ) = 1 OR RF (X, Y ) = 0
```

3) If the comparison category of UDT_x is STATE, then:

A) Let *SF* be the <routine name> of the comparison function of UDT_x.

B) `X = Y`

has the same result as

```
SF (X, Y ) = TRUE
```

C) `X <> Y`

has the same result as

```
SF (X, Y ) = FALSE
```

NOTE 180 — Rules for the comparison of user-defined types in which <comp op> is other than <equals operator> or <less than operator> are included for informational purposes only, since such predicates are equivalent to other <comparison predicate>s whose <comp op> is <equals operator> or <less than operator>.

v) Otherwise, the result of

```
X <comp op> Y
```

is *True* or *False* as follows:

1) *X* = *Y*

   is *True* if and only if *XV* and *YV* are equal.

2) *X* < *Y*

   is *True* if and only if *XV* is less than *YV*.

3) *X* <comp op> *Y*

   is *False* if and only if

   X <comp op> *Y*

   is not *True*

2) Numbers are compared with respect to their algebraic value.

3) The comparison of two character strings is determined as follows:

   a) Let *CS* be the collation as determined by Subclause 9.13, "Collation determination", for the declared types of the two character strings.

   b) If the length in characters of *X* is not equal to the length in characters of *Y*, then the shorter string is effectively replaced, for the purposes of comparison, with a copy of itself that has been extended to the length of the longer string by concatenation on the right of one or more pad characters, where the pad character is chosen based on *CS*. If *CS* has the NO PAD characteristic, then the pad character is an implementation-dependent character different from any character in the character set of *X* and *Y* that collates less than any string under *CS*. Otherwise, the pad character is a <space>.

   c) The result of the comparison of *X* and *Y* is given by the collation *CS*.

   d) Depending on the collation, two strings may compare as equal even if they are of different lengths or contain different sequences of characters. When any of the operations MAX, MIN, and DISTINCT reference a grouping column, and the UNION, EXCEPT, and INTERSECT operators refer to character strings, the specific value selected by these operations from a set of such equal values is implementation-dependent.

4) The comparison of two binary string values, *X* and *Y*, is determined by comparison of their octets with the same ordinal position. If X_i and Y_i are the values of the *i*-th octets of *X* and *Y*, respectively, and if L_x is the length in octets of *X* AND L_y is the length in octets of *Y*, then *X* is equal to *Y* if and only if $L_x = L_y$ and if $X_i = Y_i$ for all *i*.

5) The comparison of two datetimes is determined according to the interval resulting from their subtraction. Let *X* and *Y* be the two values to be compared and let *H* be the least significant <primary datetime field> of *X* and *Y*, including fractional seconds precision if the data type is time or timestamp.

   a) *X* is equal to *Y* if and only if

   ```
   ( X - Y ) INTERVAL H = INTERVAL '0' H
   ```

   is *True*.

   b) *X* is less than *Y* if and only if

   ```
   ( X - Y ) INTERVAL H < INTERVAL '0' H
   ```

is *True*.

NOTE 181 — Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.6.2, "Datetimes".

6) The comparison of two intervals is determined by the comparison of their corresponding values after conversion to integers in some common base unit. Let *X* and *Y* be the two intervals to be compared. Let *A* TO *B* be the specified or implied datetime qualifier of *X* and *C* TO *D* be the specified or implied datetime qualifier of *Y*. Let *T* be the least significant <primary datetime field> of *B* and *D* and let *U* be a datetime qualifier of the form *T*(*N*), where *N* is an <interval leading field precision> large enough so that significance is not lost in the CAST operation.

Let *XVE* be the <value expression>

```
CAST ( X AS INTERVAL U )
```

Let *YVE* be the <value expression>

```
CAST ( Y AS INTERVAL U )
```

a) *X* is equal to *Y* if and only if

```
CAST ( XVE AS INTEGER ) = CAST ( YVE AS INTEGER )
```

is *True*.

b) *X* is less than *Y* if and only if

```
CAST ( XVE AS INTEGER ) < CAST ( YVE AS INTEGER )
```

is *True*.

7) In comparisons of boolean values, *True* is greater than *False*

8) The result of comparing two reference values *X* and *Y* is determined by the comparison of their octets with the same ordinal position. Let L_x be the length in octets of *X* and let L_y be the length in octets of *Y*. Let X_i and Y_i, 1 (one) $\leq i \leq L_x$, be the values of the *i*-th octets of *X* and *Y*, respectively. *X* is equal to *Y* if and only if $L_x = L_y$ and, for all *i*, $X_i = Y_i$.

Conformance Rules

None.

NOTE 182 — If <comp op> is <equals operator> or <not equals operator>, then the Conformance Rules of Subclause 9.9, "Equality operations", apply. Otherwise, the Conformance Rules of Subclause 9.12, "Ordering operations", apply.

8.3 <between predicate>

Function

Specify a range comparison.

Format

```
<between predicate> ::= <row value predicand> <between predicate part 2>

<between predicate part 2> ::=
    [ NOT ] BETWEEN [ ASYMMETRIC | SYMMETRIC ]
    <row value predicand> AND <row value predicand>
```

Syntax Rules

1) If neither SYMMETRIC nor ASYMMETRIC is specified, then ASYMMETRIC is implicit.

2) Let *X*, *Y*, and *Z* be the first, second, and third <row value predicand>s, respectively.

3) "*X* NOT BETWEEN SYMMETRIC *Y* AND *Z*" is equivalent to "NOT (*X* BETWEEN SYMMETRIC *Y* AND *Z*)".

4) "*X* BETWEEN SYMMETRIC *Y* AND *Z*" is equivalent to "((*X* BETWEEN ASYMMETRIC *Y* AND *Z*) OR (*X* BETWEEN ASYMMETRIC *Z* AND *Y*))".

5) "*X* NOT BETWEEN ASYMMETRIC *Y* AND *Z*" is equivalent to "NOT (*X* BETWEEN ASYMMETRIC *Y* AND *Z*)".

6) "*X* BETWEEN ASYMMETRIC *Y* AND *Z*" is equivalent to "*X*>=*Y* AND *X*<=*Z*".

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T461, "Symmetric BETWEEN predicate", conforming SQL language shall not contain SYMMETRIC or ASYMMETRIC.

NOTE 183 — Since <between predicate> is an ordering operation, the Conformance Rules of Subclause 9.12, "Ordering operations", also apply.

8.4 <in predicate>

Function

Specify a quantified comparison.

Format

```
<in predicate> ::= <row value predicand> <in predicate part 2>

<in predicate part 2> ::= [ NOT ] IN <in predicate value>

<in predicate value> ::=
    <table subquery>
  | <left paren> <in value list> <right paren>

<in value list> ::= <row value expression> [ { <comma> <row value expression> }... ]
```

Syntax Rules

1) If <in value list> consists of a single <row value expression>, then that <row value expression> shall not be a <scalar subquery>.

   NOTE 184 — This Syntax Rule resolves an ambiguity in which <in predicate value> might be interperted either as a <table subquery> or as a <scalar subquery>. The ambiguity is resolved by adopting the interpretation that the <in predicate value> will be interpreted as a <table subquery>.

2) Let *IVL* be an <in value list>.

   (*IVL*)

   is equivalent to the <table value constructor>:

   (VALUES *IVL*)

3) Let *RVC* be the <row value predicand> and let *IPV* be the <in predicate value>.

4) The expression

   RVC NOT IN *IPV*

   is equivalent to

   NOT (*RVC* IN *IPV*)

5) The expression

   RVC IN *IPV*

   is equivalent to

   RVC = ANY *IPV*

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F561, "Full value expressions", conforming SQL language shall not contain a <row value expression> immediately contained in an <in value list> that is not a <value specification>.

NOTE 185 — Since <in predicate> is an equality operation, the Conformance Rules of Subclause 9.9, "Equality operations", also apply.

8.5 <like predicate>

Function

Specify a pattern-match comparison.

Format

```
<like predicate> ::=
    <character like predicate>
  | <octet like predicate>

<character like predicate> ::=
    <row value predicand> <character like predicate part 2>

<character like predicate part 2> ::=
    [ NOT ] LIKE <character pattern> [ ESCAPE <escape character> ]

<character pattern> ::= <character value expression>

<escape character> ::= <character value expression>

<octet like predicate> ::=
    <row value predicand> <octet like predicate part 2>

<octet like predicate part 2> ::=
    [ NOT ] LIKE <octet pattern> [ ESCAPE <escape octet> ]

<octet pattern> ::= <blob value expression>

<escape octet> ::= <blob value expression>
```

Syntax Rules

1) The <row value predicand> immediately contained in <character like predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared types of *CVE*, <character pattern>, and <escape character> shall be character string. *CVE*, <character pattern>, and <escape character> shall be comparable.

2) The <row value predicand> immediately contained in <octet like predicate> shall be a <row value constructor element> that is a <common value expression> *OVE*. The declared types of *OVE*, <octet pattern>, and <escape octet> shall be binary string.

3) If <character like predicate> is specified, then:

   a) Let *MC* be the <character value expression> of *CVE*, let *PC* be the <character value expression> of the <character pattern>, and let *EC* be the <character value expression> of the <escape character> if one is specified.

   b) *MC* NOT LIKE *PC*

      is equivalent to

```
NOT (MC LIKE PC)
```

c) *MC* NOT LIKE *PC* ESCAPE *EC*

is equivalent to

```
NOT (MC LIKE PC ESCAPE EC)
```

d) The collation used for <like predicate> is determined by applying Subclause 9.13, "Collation determination", with operands *CVE*, *PC*, and (if specified) *EC*.

It is implementation-defined which collations can be used as collations for the <like predicate>.

4) If <octet like predicate> is specified, then:

a) Let *MB* be the <blob value expression> of the *OVE*, let *PB* be the <blob value expression> of the <octet pattern>, and let *EB* be the <blob value expression> of the <escape octet> if one is specified.

b) *MB* NOT LIKE *PB*

is equivalent to

```
NOT (MB LIKE PB)
```

c) *MB* NOT LIKE *PB* ESCAPE *EB*

is equivalent to

```
NOT (MB LIKE PB ESCAPE EB)
```

Access Rules

None.

General Rules

1) Let *MCV* be the value of *MC* and let *PCV* be the value of *PC*. If *EC* is specified, then let *ECV* be its value.

2) Let *MBV* be the value of *MB* and let *PBV* be the value of *PB*. If *EB* is specified, then let *EBV* be its value.

3) If <character like predicate> is specified, then:

a) Case:

i) If ESCAPE is not specified and either *MCV* or *PCV* are null values, then the result of

```
MC LIKE PC
```

is *Unknown*.

ii) If ESCAPE is specified and one or more of *MCV*, *PCV* and *ECV* are null values, then the result of

MC LIKE *PC* ESCAPE *EC*

is *Unknown*.

NOTE 186 — If none of *MCV*, *PCV*, and *ECV* (if present) are null values, then the result is either *True* or *False*.

b) Case:

i) If an <escape character> is specified, then:

1) If the length in characters of *ECV* is not equal to 1, then an exception condition is raised: *data exception — invalid escape character*.

2) If there is not a partitioning of the string *PCV* into substrings such that each substring has length 1 (one) or 2, no substring of length 1 (one) is the escape character *ECV*, and each substring of length 2 is the escape character *ECV* followed by either the escape character *ECV*, an <underscore> character, or the <percent> character, then an exception condition is raised: *data exception — invalid escape sequence*.

If there is such a partitioning of *PCV*, then in that partitioning, each substring with length 2 represents a single occurrence of the second character of that substring and is called a *single character specifier*.

Each substring with length 1 (one) that is the <underscore> character represents an *arbitrary character specifier*. Each substring with length 1 (one) that is the <percent> character represents an *arbitrary string specifier*. Each substring with length 1 (one) that is neither the <underscore> character nor the <percent> character represents the character that it contains and is called a *single character specifier*.

ii) If an <escape character> is not specified, then each <underscore> character in *PCV* represents an arbitrary character specifier, each <percent> character in *PCV* represents an arbitrary string specifier, and each character in *PCV* that is neither the <underscore> character nor the <percent> character represents itself and is called a *single character specifier*.

c) Case:

i) If *MCV* and *PCV* are character strings whose lengths are variable and if the lengths of both *MCV* and *PCV* are 0 (zero), then

MC LIKE *PC*

is *True*.

ii) The <predicate>

MC LIKE *PC*

is *True* if there exists a partitioning of *MCV* into substrings such that:

1) A substring of *MCV* is a sequence of 0 (zero) or more contiguous characters of *MCV* and each character of *MCV* is part of exactly one substring.

2) If the *i*-th substring of *PCV* is an arbitrary character specifier, then the *i*-th substring of *MCV* is any single character.

3) If the *i*-th substring of *PCV* is an arbitrary string specifier, then the *i*-th substring of *MCV* is any sequence of 0 (zero) or more characters.

4) If the *i*-th substring of *PCV* is a single character specifier, then the *i*-th substring of *MCV* contains exactly 1 (one) character that is equal to the character represented by the single character specifier according to the collation of the <like predicate>.

5) The number of substrings of *MCV* is equal to the number of substring specifiers of *PCV*.

iii) Otherwise,

MC LIKE *PC*

is *False*.

4) If <octet like predicate> is specified, then:

a) Case:

i) If ESCAPE is not specified and either *MBV* or *PBV* are null values, then the result of

MB LIKE *PB*

is *Unknown*.

ii) If ESCAPE is specified and one or more of *MBV*, *PBV* and *EBV* are null values, then the result of

MB LIKE *PB* ESCAPE *EB*

is *Unknown*.

NOTE 187 — If none of *MBV*, *PBV*, and *EBV* (if present) are null values, then the result is either *True* or *False*.

b) <percent> in the context of an <octet like predicate> has the same bit pattern as the encoding of a <percent> in the SQL_TEXT character set.

c) <underscore> in the context of an <octet like predicate> has the same bit pattern as the encoding of an <underscore> in the SQL_TEXT character set.

d) Case:

i) If an <escape octet> is specified, then:

1) If the length in octets of *EBV* is not equal to 1, then an exception condition is raised: *data exception — invalid escape octet.*

2) If there is not a partitioning of the string *PBV* into substrings such that each substring has length 1 (one) or 2, no substring of length 1 (one) is the escape octet *EBV*, and each substring of length 2 is the escape octet *EBV* followed by either the escape octet *EBV*, an <underscore> octet, or the <percent> octet, then an exception condition is raised: *data exception — invalid escape sequence.*

If there is such a partitioning of *PBV*, then in that partitioning, each substring with length 2 represents a single occurrence of the second octet of that substring. Each substring with length 1 (one) that is the <underscore> octet represents an *arbitrary octet specifier*. Each substring with length 1 (one) that is the <percent> octet represents an *arbitrary string*

specifier. Each substring with length 1 (one) that is neither the <underscore> octet nor the <percent> octet represents the octet that it contains.

ii) If an <escape octet> is not specified, then each <underscore> octet in *PBV* represents an arbitrary octet specifier, each <percent> octet in *PBV* represents an arbitrary string specifier, and each octet in *PBV* that is neither the <underscore> octet nor the <percent> octet represents itself.

e) The string *PBV* is a sequence of the minimum number of substring specifiers such that each portion of *PBV* is part of exactly one substring specifier. A *substring specifier* is an arbitrary octet specifier, and arbitrary string specifier, or any sequence of octets other than an arbitrary octet specifier or an arbitrary string specifier.

f) Case:

i) If the lengths of both *MBV* and *PBV* are 0 (zero), then

MB LIKE *PB*

is *True*.

ii) The <predicate>

MB LIKE *PB*

is *True* if there exists a partitioning of *MBV* into substrings such that:

1) A substring of *MBV* is a sequence of 0 (zero) or more contiguous octets of *MBV* and each octet of *MBV* is part of exactly one substring.

2) If the *i*-th substring specifier of *PBV* is an arbitrary octet specifier, the *i*-th substring of *MBV* is any single octet.

3) the *i*-th substring specifier of *PBV* is an arbitrary string specifier, then the *i*-th substring of *MBV* is any sequence of 0 (zero) or more octets.

4) If the *i*-th substring specifier of *PBV* is an neither an arbitrary character specifier not an arbitrary string specifier, then the *i*-th substring of *MBV* has the same length and bit pattern as that of the substring specifier.

5) The number of substrings of *MBV* is equal to the number of substring specifiers of *PBV*.

iii) Otherwise:

MB LIKE *PB*

is *False*.

Conformance Rules

1) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain an <octet like predicate>.

2) Without Feature F281, "LIKE enhancements", conforming SQL language shall not contain a <common value expression> simply contained in the <row value predicand> immediately contained in <character like predicate> that is not a column reference.

3) Without Feature F281, "LIKE enhancements", conforming SQL language shall not contain a <character pattern> that is not a <value specification>.

4) Without Feature F281, "LIKE enhancements", conforming SQL language shall not contain an <escape character> that is not a <value specification>.

5) Without Feature T042, "Extended LOB data type support", in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type CHARACTER LARGE OBJECT

6) Without Feature F421, "National character", and Feature T042, "Extended LOB data type support", in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.

8.6 <similar predicate>

Function

Specify a character string similarity by means of a regular expression.

Format

```
<similar predicate> ::=
    <row value predicand> <similar predicate part 2>

<similar predicate part 2> ::=
    [ NOT ] SIMILAR TO <similar pattern> [ ESCAPE <escape character> ]

<similar pattern> ::= <character value expression>

<regular expression> ::=
    <regular term>
  | <regular expression> <vertical bar> <regular term>

<regular term> ::=
    <regular factor>
  | <regular term> <regular factor>

<regular factor> ::=
    <regular primary>
  | <regular primary> <asterisk>
  | <regular primary> <plus sign>
  | <regular primary> <question mark>
  | <regular primary> <repeat factor>

<repeat factor> ::= <left brace> <low value> [ <upper limit> ] <right brace>

<upper limit> ::= <comma> [ <high value> ]

<low value> ::= <unsigned integer>

<high value> ::= <unsigned integer>

<regular primary> ::=
    <character specifier>
  | <percent>
  | <regular character set>
  | <left paren> <regular expression> <right paren>

<character specifier> ::=
    <non-escaped character>
  | <escaped character>

<non-escaped character> ::= !! See the Syntax Rules

<escaped character> ::= !! See the Syntax Rules

<regular character set> ::=
    <underscore>
```

```
  | <left bracket> <character enumeration>... <right bracket>
  | <left bracket> <circumflex> <character enumeration>... <right bracket>
  | <left bracket> <character enumeration include>...
    <circumflex> <character enumeration exclude>... <right bracket>

<character enumeration include> ::= <character enumeration>

<character enumeration exclude> ::= <character enumeration>

<character enumeration> ::=
    <character specifier>
  | <character specifier> <minus sign> <character specifier>
  | <left bracket> <colon> <regular character set identifier> <colon> <right bracket>

<regular character set identifier> ::= <identifier>
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared types of *CVE*, <similar pattern>, and <escape character> shall be character string. *CVE*, <similar pattern>, and <escape character> shall be comparable.

2) Let *CM* be the <character value expression> of *CVE* and let *SP* be the <similar pattern>. If <escape character> *EC* is specified, then

   CM NOT SIMILAR TO *SP* ESCAPE *EC*

   is equivalent to

   NOT (*CM* SIMILAR TO *SP* ESCAPE *EC*)

   If <escape character> *EC* is not specified, then

   CM NOT SIMILAR TO *SP*

   is equivalent to

   NOT (*CM* SIMILAR TO *SP*)

3) The value of the <identifier> that is a <regular character set identifier> shall be either ALPHA, UPPER, LOWER, DIGIT, ALNUM, SPACE, or WHITESPACE.

4) The collation used for <similar predicate> is determined by applying Subclause 9.13, "Collation determination", with operands *CVE*, *PC*, and (if specified) *EC*.

   It is implementation-defined which collations can be used as collations for <similar predicate>.

5) A <non-escaped character> is any single character from the character set of the <similar pattern> that is not a <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>, or the character specified by the result of the <character value expression> of <escape character>. A <character specifier> that is a <non-escaped character> represents itself.

6) An <escaped character> is a sequence of two characters: the character specified by the result of the <character value expression> of <escape character>, followed by a second character that is a <left bracket>,

<right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>, or the character specified by the result of the <character value expression> of <escape character>. A <character specifier> that is an <escaped character> represents its second character.

7) The value of <low value> shall be a positive integer. The value of <high value> shall be greater than or equal to the value of <low value>.

Access Rules

None.

General Rules

1) Let *MCV* be the result of the <character value expression> of *CVE* and let *PCV* be the result of the <character value expression> of the <similar pattern>. If *EC* is specified, then let *ECV* be its value.

2) If the result of the <character value expression> of the <similar pattern> is not a zero-length string and does not have the format of a <regular expression>, then an exception condition is raised: *data exception — invalid regular expression.*

3) If an <escape character> is specified, then:

   a) If the length in characters of *ECV* is not equal to 1 (one), then an exception condition is raised: *data exception — invalid escape character.*

   b) If *ECV* is one of <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, or <left brace> and *ECV* occurs in the <regular expression> except in an <escaped character>, then an exception condition is raised: *data exception — invalid use of escape character.*

   c) If *ECV* is a <colon> and the <regular expression> contains a <regular character set identifier>, then an exception condition is raised: *data exception — escape character conflict.*

4) Case:

   a) If ESCAPE is not specified, then if either or both of *MCV* and *PCV* are the null value, then the result of

      CM SIMILAR TO *SP*

      is *Unknown*.

   b) If ESCAPE is specified, then if one or more of *MCV*, *PCV*, and *ECV* are the null value, then the result of

      CM SIMILAR TO *SP* ESCAPE *EC*

      is *Unknown*.

   NOTE 188 — If none of *MCV*, *PCV*, and *ECV* (if present) are the null value, then the result is either *True* or *False*.

5) The set of characters in a <character enumeration> is defined as

a) If the enumeration is specified in the form "<character specifier> <minus sign> <character specifier>", then the set of all characters that collate greater than or equal to the character represented by the left <character specifier> and less than or equal to the character represented by the right <character specifier>, according to the collation of the pattern *PCV*.

b) Otherwise, the character that the <character specifier> in the <character enumeration> represents.

6) Let *LV* be the value of the <low value> contained in a <repeat factor> *RF*.

Case:

a) If *RF* does not contain an <upper limit>, then let *HV* be *LV*.

b) If *RF* contains an <upper limit> that contains a <high value>, then let *HV* be the value of <high value>.

c) Otherwise, let *HV* be the length or maximum length of *CVE*.

7) Let *R* be the result of the <character value expression> of the <similar pattern>. The regular language L(*R*) of the <similar pattern> is a (possibly infinite) set of strings. It is defined recursively for well-formed <regular expression>s *Q*, *Q1*, and *Q2* by the following rules:

a) L(*Q1* <vertical bar> *Q2*)

is the union of L(*Q1*) and L(*Q2*)

b) L(*Q* <asterisk>)

is the set of all strings that can be constructed by concatenating zero or more strings from L(*Q*).

c) L(*Q* <plus sign>)

is the set of all strings that can be constructed by concatenating one or more strings from L(*Q*).

d) L(*Q* <repeat factor>)

is the set of all strings that can be constructed by concatenating *NS*, $LV \le NS \le HV$, strings from L(*Q*).

e) L(<character specifier>)

is a set that contains a single string of length 1 (one) with the character that the <character specifier> represents

f) L(<percent>)

is the set of all strings of any length (zero or more) from the character set of the pattern *PCV*.

g) L(<question mark>)

is the set of all strings that can be constructed by concatenating exactly 0 (zero) or 1 (one) strings from L(*Q*).

h) L(<left paren> *Q* <right paren>)

is equal to L(*Q*)

i) L(<underscore>)

is the set of all strings of length 1 (one) from the character set of the pattern *PCV*.

j) L(<left bracket> <character enumeration> <right bracket>)

is the set of all strings of length 1 (one) from the set of characters in the <character enumeration>s.

k) L(<left bracket> <circumflex> <character enumeration> <right bracket>)

is the set of all strings of length 1 (one) with characters from the character set of the pattern *PCV* that are not contained in the set of characters in the <character enumeration>.

l) L(<left bracket> <character enumeration include> <circumflex> <character enumeration exclude> <right bracket>)

is the set of all strings of length 1 (one) taken from the set of characters in the <character enumeration include>s, except for those strings of length 1 (one) taken from the set of characters in the <character enumeration exclude>.

m) L(<left bracket> <colon> ALPHA <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin letter>s.

n) L(<left bracket> <colon> UPPER <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin upper case letter>s.

o) L(<left bracket> <colon> LOWER <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin lower case letter>s.

p) L(<left bracket> <colon> DIGIT <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <digit>s.

q) L(<left bracket> <colon> SPACE <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are the <space> character.

r) L(<left bracket> <colon> WHITESPACE <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are white space characters.

NOTE 189 — "white space" is defined in Subclause 3.1.6, "Definitions provided in Part 2".

s) L(<left bracket> <colon> ALNUM <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin letter>s or <digit>s.

t) `L( Q1 || Q2 )`

is the set of all strings that can be constructed by concatenating one element of L(*Q1*) and one element of L(*Q2*).

u) L(*Q*)

is the set of the zero-length string, if *Q* is an empty regular expression.

8) The <similar predicate>

CM SIMILAR TO *SP*

is *True*, if there exists at least one element *X* of L(*R*) that is equal to *MCV* according to the collation of the <similar predicate>; otherwise, it is *False*.

NOTE 190 — The <similar predicate> is defined differently from equivalent forms of the LIKE predicate. In particular, blanks at the end of a pattern and collation are handled differently.

Conformance Rules

1) Without Feature T141, "SIMILAR predicate", conforming SQL language shall not contain a <similar predicate>.

2) Without Feature T042, "Extended LOB data type support", in conforming SQL language, a <character value expression> simply contained in a <similar predicate> shall not be of declared type CHARACTER LARGE OBJECT.

8.7 <null predicate>

Function

Specify a test for a null value.

Format

```
<null predicate> ::= <row value predicand> <null predicate part 2>

<null predicate part 2> ::= IS [ NOT ] NULL
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *R* be the value of the <row value predicand>.

2) Case:

   a) If *R* is the null value, then "`R IS NULL`" is *True*.

   b) Otherwise:

      i) The value of "`R IS NULL`" is

         Case:

         1) If the value of every field in *R* is the null value, then *True*.

         2) Otherwise, *False*.

      ii) The value of "`R IS NOT NULL`" is

         Case:

         1) If the value of no field in *R* is the null value, then *True*.

         2) Otherwise, *False*.

NOTE 191 — For all *R*, "*R* IS NOT NULL" has the same result as "NOT *R* IS NULL" if and only if *R* is of degree 1. Table 14, "<null predicate> semantics", specifies this behavior.

Table 14 — <null predicate> semantics

Expression	*R* IS NULL	*R* IS NOT NULL	NOT *R* IS NULL	NOT *R* IS NOT NULL
degree 1: null	*True*	*False*	*False*	*True*
degree 1: not null	*False*	*True*	*True*	*False*
degree > 1: all null	*True*	*False*	*False*	*True*
degree > 1: some null	*False*	*False*	*True*	*True*
degree > 1: none null	*False*	*True*	*True*	*False*

Conformance Rules

None.

8.8 <quantified comparison predicate>

Function

Specify a quantified comparison.

Format

```
<quantified comparison predicate> ::=
    <row value predicand> <quantified comparison predicate part 2>

<quantified comparison predicate part 2> ::=
    <comp op> <quantifier> <table subquery>

<quantifier> ::=
    <all>
  | <some>

<all> ::= ALL

<some> ::=
    SOME
  | ANY
```

Syntax Rules

1) Let *RV1* and *RV2* be <row value predicand>s whose declared types are respectively that of the <row value predicand> and the row type of the <table subquery>. The Syntax Rules of Subclause 8.2, "<comparison predicate>", are applied to:

   RV1 <comp op> *RV2*

Access Rules

None.

General Rules

1) Let *R* be the result of the <row value predicand> and let *T* be the result of the <table subquery>.

2) The result of "*R* <comp op> <quantifier> *T*" is derived by the application of the implied <comparison predicate> "*R* <comp op> *RT*" to every row *RT* in *T*:

   Case:

   a) If *T* is empty or if the implied <comparison predicate> is *True* for every row *RT* in *T*, then "*R* <comp op> <all> *T*" is *True*.

b) If the implied <comparison predicate> is *False* for at least one row *RT* in *T*, then "*R* <comp op> <all> *T*" is *False*.

c) If the implied <comparison predicate> is *True* for at least one row *RT* in *T*, then "*R* <comp op> <some> *T*" is *True*.

d) If *T* is empty or if the implied <comparison predicate> is *False* for every row *RT* in *T*, then "*R* <comp op> <some> *T*" is *False*.

e) If "*R* <comp op> <quantifier> *T*" is neither *True* nor *False*, then it is *Unknown*.

Conformance Rules

None.

NOTE 192 — If <equals operator> or <not equals operator> is specified, then the <quantified comparison predicate> is an equality operator and the Conformance Rules of Subclause 9.9, "Equality operations", apply. Otherwise, the <quantified comparison predicate> is an ordering operation, and the Conformance Rules of Subclause 9.12, "Ordering operations", apply.

8.9 <exists predicate>

Function

Specify a test for a non-empty set.

Format

```
<exists predicate> ::= EXISTS <table subquery>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *T* be the result of the <table subquery>.

2) If the cardinality of *T* is greater than 0 (zero), then the result of the <exists predicate> is *True*; otherwise, the result of the <exists predicate> is *False*.

Conformance Rules

1) Without Feature T501, "Enhanced EXISTS predicate", conforming SQL language shall not contain an <exists predicate> that simply contains a <table subquery> in which the <select list> of a <query specification> directly contained in the <table subquery> does not comprise either an <asterisk> or a single <derived column>.

8.10 <unique predicate>

Function

Specify a test for the absence of duplicate rows.

Format

```
<unique predicate> ::= UNIQUE <table subquery>
```

Syntax Rules

1) Each column of user-defined type in the result of the <table subquery> shall have a comparison type.

2) Each column of the <table subquery> is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

Access Rules

None.

General Rules

1) Let *T* be the result of the <table subquery>.

2) If there are no two rows in *T* such that the value of each column in one row is non-null and is not distinct from the value of the corresponding column in the other row, then the result of the <unique predicate> is *True*; otherwise, the result of the <unique predicate> is *False*.

Conformance Rules

1) Without Feature F291, "UNIQUE predicate", conforming SQL language shall not contain a <unique predicate>.

NOTE 193 — The Conformance Rules of Subclause 9.10, "Grouping operations", also apply.

8.11 <normalized predicate>

Function

Determine whether a character string value is normalized.

Format

```
<normalized predicate> ::= <row value predicand> <normalized predicate part 2>

<normalized predicate part 2> ::= IS [ NOT ] NORMALIZED
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared type of *CVE* shal be character string and the character set of *CVE* shall be UTF8, UTF16, or UTF32.

2) The expression
   CVE IS NOT NORMALIZED
   is equivalent to
   NOT (*CVE* IS NORMALIZED)

Access Rules

None.

General Rules

1) The result of *CVE* IS NORMALIZED is

   Case:

   a) If the value of *CVE* is the null value, then *Unknown*.

   b) Otherwise, if the value of *CVE* is in Normalization Form C, as specified by [Unicode15], then *True*; otherwise, *False*.

Conformance Rules

1) Without Feature T061, "UCS support", conforming SQL language shall not contain a <normalized predicate>.

8.12 <match predicate>

Function

Specify a test for matching rows.

Format

```
<match predicate> ::= <row value predicand> <match predicate part 2>

<match predicate part 2> ::=
    MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ] <table subquery>
```

Syntax Rules

1) The row type of the <row value predicand> and the row type of the <table subquery> shall be comparable.

2) Each field of <row value predicand> and each column of <table subquery> is an operand of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

3) If neither SIMPLE, PARTIAL, nor FULL is specified, then SIMPLE is implicit.

Access Rules

None.

General Rules

1) Let *R* be the <row value predicand>.

2) If SIMPLE is specified or implicit, then

   Case:

   a) If *R* is the null value, then the <match predicate> is *True*.

   b) Otherwise:

      i) If the value of some field in *R* is the null value, then the <match predicate> is *True*.

      ii) If the value of no field in *R* is the null value, then

          Case:

          1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that

             $$R = RT_i$$

             then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that

$R = RT_i$

then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

3) If PARTIAL is specified, then

Case:

a) If R is the null value, then the <match predicate> is *True*.

b) Otherwise,

Case:

i) If the value of every field in R is the null value, then the <match predicate> is *True*.

ii) Otherwise,

Case:

1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that each non-null value of R equals its corresponding value in RT_i, then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that each non-null value of R equals its corresponding value in RT_i, then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

4) If FULL is specified, then

Case:

a) If R is the null value, then the <match predicate> is *True*.

b) Otherwise,

Case:

i) If the value of every field in R is the null value, then the <match predicate> is *True*.

ii) If the value of no field in R is the null value, then

Case:

1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that

$R = RT_i$

then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that

$$R = RT_i$$

then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

iii) Otherwise, the <match predicate> is *False*.

Conformance Rules

1) Without Feature F741, "Referential MATCH types", conforming SQL language shall not contain a <match predicate>.

NOTE 194 — The Conformance Rules of Subclause 9.9, "Equality operations", also apply.

8.13 <overlaps predicate>

Function

Specify a test for an overlap between two datetime periods.

Format

```
<overlaps predicate> ::= <overlaps predicate part 1> <overlaps predicate part 2>

<overlaps predicate part 1> ::= <row value predicand 1>

<overlaps predicate part 2> ::= OVERLAPS <row value predicand 2>

<row value predicand 1> ::= <row value predicand>

<row value predicand 2> ::= <row value predicand>
```

Syntax Rules

1) The degrees of <row value predicand 1> and <row value predicand 2> shall both be 2.

2) The declared types of the first field of <row value predicand 1> and the first field of <row value predicand 2> shall both be datetime data types and these data types shall be comparable.

   NOTE 195 — Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.6.2, "Datetimes".

3) The declared type of the second field of each <row value predicand> shall be a datetime data type or INTERVAL.

   Case:

   a) If the declared type is INTERVAL, then the precision of the declared type shall be such that the interval can be added to the datetime data type of the first column of the <row value predicand>.

   b) If the declared type is a datetime data type, then it shall be comparable with the datetime data type of the first column of the <row value predicand>.

Access Rules

None.

General Rules

1) If the value of <row value predicand 1> is the null value or the value of <row value predicand 2> is the null value, then the result of the <overlaps predicate> is *Unknown* and no further General Rules of this Subclause are applied.

2) Let *D1* be the value of the first field of <row value predicand 1> and *D2* be the value of the first field of <row value predicand 2>.

3) Case:

   a) If the most specific type of the second field of <row value predicand 1> is a datetime data type, then let *E1* be the value of the second field of <row value predicand 1>.

   b) If the most specific type of the second field of <row value predicand 1> is INTERVAL, then let *I1* be the value of the second field of <row value predicand 1>. Let *E1* = *D1* + *I1*.

4) If *D1* is the null value or if *E1* < *D1*, then let *S1* = *E1* and let *T1* = *D1*. Otherwise, let *S1* = *D1* and let *T1* = *E1*.

5) Case:

   a) If the most specific type of the second field of <row value predicand 2> is a datetime data type, then let *E2* be the value of the second field of <row value predicand 2>.

   b) If the most specific type of the second field of <row value predicand 2> is INTERVAL, then let *I2* be the value of the second field of <row value predicand 2>. Let *E2* = *D2* + *I2*.

6) If *D2* is the null value or if *E2* < *D2*, then let *S2* = *E2* and let *T2* = *D2*. Otherwise, let *S2* = *D2* and let *T2* = *E2*.

7) The result of the <overlaps predicate> is the result of the following expression:

```
( S1 > S2 AND NOT ( S1 >= T2 AND T1 >= T2 ) )
    OR
( S2 > S1 AND NOT ( S2 >= T1 AND T2 >= T1 ) )
    OR
( S1 = S2 AND ( T1 <> T2 OR T1 = T2 ) )
```

Conformance Rules

1) Without Feature F053, "OVERLAPS predicate", conforming SQL language shall not contain an <overlaps predicate>.

8.14 <distinct predicate>

Function

Specify a test of whether two row values are distinct

Format

```
<distinct predicate> ::=
    <row value predicand 3> <distinct predicate part 2>

<distinct predicate part 2> ::=
    IS [ NOT ] DISTINCT FROM <row value predicand 4>

<row value predicand 3> ::= <row value predicand>

<row value predicand 4> ::= <row value predicand>
```

Syntax Rules

1) The two <row value predicand>s shall be of the same degree.

2) Let *respective values* be values with the same ordinal position.

3) The declared types of the respective values of the two <row value predicand>s shall be comparable.

4) Let *X* be the first <row value predicand> and let *Y* be the second <row value predicand>.

5) Each field of each <row value predicand> is an operand of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

6) If <distinct predicate part 2> immediately contains NOT, then the <distinct predicate> is equivalent to:

```
NOT ( X IS DISTINCT FROM Y )
```

Access Rules

None.

General Rules

1) The result of <distinct predicate> is *True* if the value of <row value predicand 3> is distinct from the value of <row value predicand 4>; otherwise, the result is *False*.

NOTE 196 — "distinct" is defined in Subclause 3.1.6, "Definitions provided in Part 2".

2) If two <row value predicand>s are not distinct, then they are said to be *duplicates*. If a number of <row value predicand>s are all duplicates of each other, then all except one are said to be *redundant duplicates*.

Conformance Rules

1) Without Feature T151, "DISTINCT predicate", conforming SQL language shall not contain a <distinct predicate>.

   NOTE 197 — The Conformance Rules of Subclause 9.9, "Equality operations", also apply.

2) Without Feature T152, "DISTINCT predicate with negation", conforming SQL language shall not contain a <distinct predicate part 2> that immediately contains NOT.

8.15 <member predicate>

Function

Specify a test of whether a value is a member of a multiset.

Format

```
<member predicate> ::=
    <row value predicand> <member predicate part 2>

<member predicate part 2> ::=
    [ NOT ] MEMBER [ OF ] <multiset value expression>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression> and let *ET* be the declared element type of *MVE*.

2) Case:

   a) If the <row value predicand> is a <row value constructor predicand> that is a single <common value expression> or <boolean value expression> *CVE*, then let *X* be *CVE*.

   b) Otherwise, let *X* be the <row value predicand>.

3) The declared type of *X* shall be comparable to *ET*.

4) *X* is an operand of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

5) If <member predicate part 2> immediately contains NOT, then the <member predicate> is equivalent to

```
NOT ( X MEMBER OF MVE )
```

Access Rules

None.

General Rules

1) Let *XV* be the value of *X*, and let *MV* be the value of *MVE*.

2) Let *N* be the result of `CARDINALITY (MVE)`.

3) The <member predicate>

```
XV MEMBER OF MVE
```

is evaluated as follows:

Case:

a) If N is 0 (zero), then the <member predicate> is *False*.

b) If either XV or MV is the null value, then the <member predicate> is *Unknown*.

c) Otherwise, let ME_i for 1 (one) $\leq i \leq N$ be an enumeration of the elements of MV.

Case:

i) If $CV = ME_i$ for some i, then the <member predicate> is *True*.

ii) If ME_i is the null value for some i, then the <member predicate> is *Unknown*.

iii) Otherwise, the <member predicate> is *False*.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <member predicate>.

NOTE 198 — The Conformance Rules of Subclause 9.9, "Equality operations", also apply.

8.16 <submultiset predicate>

Function

Specify a test of whether a multiset is a submultiset of another multiset.

Format

```
<submultiset predicate> ::=
    <row value predicand> <submultiset predicate part 2>

<submultiset predicate part 2> ::=
    [ NOT ] SUBMULTISET [ OF ] <multiset value expression>
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor> that is a single <common value expression> *CVE*. The declared type of *CVE* shall be a multiset type. Let *CVET* be the declared element type of *CVE*.

2) Let *MVE* be the <multiset value expression>. Let *MVET* be the declared element type of *MVE*.

3) *CVET* shall be comparable to *MVET*.

4) *CVE* and *MVE* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

5) If <submultiset predicate part 2> immediately contains NOT, then the <member predicate> is equivalent to

```
NOT ( CVE SUBMULTISET OF MVE )
```

Access Rules

None.

General Rules

1) Let *CV* be the value of *CVE*, and let *MV* be the value of *MVE*.

2) Let *M* be the result of CARDINALITY (*CV*), and let *N* be the result of CARDINALITY (*MV*).

3) The <submultiset predicate>

```
CVE SUBMULTISET OF MVE
```

is evaluated as follows:

Case:

a) If M is 0 (zero), then the <submultiset predicate> is *True*.

b) If either CV or MV is the null value, then the <submultiset predicate> is *Unknown*.

c) Otherwise,

Case:

i) If $M > N$, then the <submultiset predicate> is *False*.

ii) If there exist an enumeration CE_i for 1 (one) $\leq i \leq M$ of the elements of CE and an enumeration ME_j for 1 (one) $\leq j \leq N$ of the elements of MV such that for all i, 1 (one) $\leq i \leq M$, $CE_i = ME_i$, then the <submultiset predicate> is *True*.

iii) If there exist an enumeration CE_i for 1 (one) $\leq i \leq M$ of the elements of CE and an enumeration ME_i for 1 (one) $\leq i \leq N$ of the elements of MV such that for all i, 1 (one) $\leq i \leq M$, $CE_i = ME_i$ is either *True* or *Unknown*, then the <submultiset predicate> is *Unknown*.

iv) Otherwise, the <submultiset predicate> is *False*.

Conformance Rules

1) Without Feature S275, "Advanced multiset support", conforming SQL language shall not contain a <submultiset predicate>.

NOTE 199 — The Conformance Rules of Subclause 9.11, "Multiset element grouping operations", also apply.

8.17 <set predicate>

Function

Specify a test of whether a multiset is a set (that is, does not contain any duplicates).

Format

```
<set predicate> ::= <row value predicand> <set predicate part 2>

<set predicate part 2> ::= IS [ NOT ] A SET
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a single <common value expression> *CVE*. The declared type of *CVE* shall be a multiset type. Let *CVET* be the element type of *CVE*.

2) *CVE* is an operand of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

3) If <set predicate part 2> immediately contains NOT, then the <set predicate> is equivalent to

```
NOT ( CVE IS A SET )
```

4) If <set predicate part 2> does not immediately contain NOT, then the <set predicate> is equivalent to

```
CARDINALITY ( CVE ) = CARDINALITY ( SET ( CVE ) )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <set predicate>.

NOTE 200 — The Conformance Rules of Subclause 9.11, "Multiset element grouping operations", also apply.

8.18 <type predicate>

Function

Specify a type test.

Format

```
<type predicate> ::=
    <row value predicand> <type predicate part 2>

<type predicate part 2> ::=
    IS [ NOT ] OF <left paren> <type list> <right paren>

<type list> ::=
    <user-defined type specification>
    [ { <comma> <user-defined type specification> }... ]

<user-defined type specification> ::=
    <inclusive user-defined type specification>
  | <exclusive user-defined type specification>

<inclusive user-defined type specification> ::=
    <path-resolved user-defined type name>

<exclusive user-defined type specification> ::=
    ONLY <path-resolved user-defined type name>
```

Syntax Rules

1) The <row value predicand> immediately contained in <type predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE*.

2) The declared type of *CVE* shall be a user-defined type.

3) For each <user-defined type name> *UDTN* contained in a <user-defined type specification>, the schema identified by the implicit or explicit schema name of *UDTN* shall include a user-defined type descriptor whose name is equivalent to the <qualified identifier> of *UDTN*.

4) Let the term *specified type* refer to a user-defined type that is specified by a <user-defined type name> contained in a <user-defined type specification>. A type specified by an <inclusive user-defined type specification> is *inclusively specified*; a type specified by an <exclusive user-defined type specification> is *exclusively specified*.

5) Let *T* be the type specified by <inclusive user-defined type specification> or <exclusive user-defined type specification>. *T* shall be a subtype of the declared type of *CVE*.

   NOTE 201 — The term "subtype family" is defined in Subclause 4.7.5, "Subtypes and supertypes". If *T1* is a member of the subtype family of *T2*, then it follows that the subtype family of *T1* and the subtype family of *T2* are the same set of types.

6) Let *TL* be the <type list>.

7) A <type predicate> of the form

CVE IS NOT OF (*TL*)

is equivalent to

NOT (*CVE* IS OF (*TL*))

Access Rules

None.

General Rules

1) Let *V* be the result of evaluating the <row value predicand>.

2) Let *ST* be the set consisting of every type that is either some exclusively specified type, or a subtype of some inclusively specified type.

3) Let *TPR* be the result of evaluating the <type predicate>.

   Case:

   a) If *V* is the null value, then *TPR* is *Unknown*.

   b) If the most specific type of *V* is a member of *ST*, then *TPR* is *True*.

   c) Otherwise, *TPR* is *False*.

Conformance Rules

1) Without Feature S151, "Type predicate", conforming SQL language shall not contain a <type predicate>.

8.19 <search condition>

Function

Specify a condition that is *True*, *False*, or *Unknown*, depending on the value of a <boolean value expression>.

Format

```
<search condition> ::= <boolean value expression>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) When a <search condition> *S* is evaluated against a row of a table, each reference to a column of that table by a column reference directly contained in *S* is a reference to the value of that column in that row.

2) The result of the <search condition> is the result of the <boolean value expression>.

Conformance Rules

None.