17 Connection management

17.1 <connect statement>

Function

Establish an SQL-session.

Format

```
<connect statement> ::= CONNECT TO <connection target>

<connection target> ::=
    <SQL-server name> [ AS <connection name> ] [ USER <connection user name> ]
  | DEFAULT
```

Syntax Rules

1) If <connection user name> is not specified, then an implementation-defined <connection user name> for the SQL-connection is implicit.

Access Rules

None.

General Rules

1) If a <connect statement> is executed after the first transaction-initiating SQL-statement executed by the current SQL-transaction and the SQL-implementation does not support transactions that affect more than one SQL-server, then an exception condition is raised: *feature not supported — multiple server transactions.*

2) If <connection user name> is specified, then let *S* be <connection user name> and let *V* be the character string that is the value of

```
TRIM ( BOTH ' ' FROM S )
```

3) If *V* does not conform to the Format and Syntax Rules of a <user identifier>, then an exception condition is raised: *invalid authorization specification.*

4) If the SQL-client module that contains the <externally-invoked procedure> that contains the <connect statement> specifies a <module authorization identifier>, then whether or not <connection user name> shall be identical to that <module authorization identifier> is implementation-defined, as are any other

restrictions on the value of <connection user name>. Otherwise, any restrictions on the value of <connection user name> are implementation-defined.

5) If the value of <connection user name> does not conform to the implementation-defined restrictions, then an exception condition is raised: *invalid authorization specification.*

6) If <connection name> was specified, then let *CV* be <simple value specification> immediately contained in <connection name>. If neither DEFAULT nor <connection name> were specified, then let *CV* be <SQL-server name>. Let *CN* be the result of

```
TRIM ( BOTH ' ' FROM CV )
```

If *CN* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid connection name.*

7) If an SQL-connection with name *CN* has already been established by the current SQL-agent and has not been disconnected, or if DEFAULT is specified and a default SQL-connection has already been established by the current SQL-agent and has not been disconnected, then an exception condition is raised: *connection exception — connection name in use.*

8) Case:

   a) If DEFAULT is specified, then the default SQL-session is initiated and associated with the default SQL-server. The method by which the default SQL-server is determined is implementation-defined.

   b) Otherwise, an SQL-session is initiated and associated with the SQL-server identified by <SQL-server name>. The method by which <SQL-server name> is used to determine the appropriate SQL-server is implementation-defined.

9) If the <connect statement> successfully initiates an SQL-session, then:

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

      NOTE 419 — The SQL-session context is defined in Subclause 4.37, "SQL-sessions".

   b) The SQL-session initiated by the <connect statement> becomes the current SQL-session and the SQL-connection established to that SQL-session becomes the current SQL-connection.

NOTE 420 — If the <connect statement> fails to initiate an SQL-session, then the current SQL-connection and current SQL-session, if any, remain unchanged.

10) If the SQL-client cannot establish the SQL-connection, then an exception condition is raised: *connection exception — SQL-client unable to establish SQL-connection.*

11) If the SQL-server rejects the establishment of the SQL-connection, then an exception condition is raised: *connection exception — SQL-server rejected establishment of SQL-connection.*

12) The SQL-server for the subsequent execution of <externally-invoked procedure>s in any SQL-client modules associated with the SQL-agent is set to the SQL-server identified by <SQL-server name>.

13) In the context of the newly established SQL-session, the authorization stack is initialized with a single cell containing the user identifier <connection user name>.

14) A new savepoint level is established.

Conformance Rules

1) Without Feature F771, "Connection management", conforming SQL language shall not contain a <connect statement>.

17.2 <set connection statement>

Function

Select an SQL-connection from the available SQL-connections.

Format

```
<set connection statement> ::= SET CONNECTION <connection object>

<connection object> ::=
    DEFAULT
  | <connection name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) If a <set connection statement> is executed after the first transaction-initiating SQL-statement executed by the current SQL-transaction and the SQL-implementation does not support transactions that affect more than one SQL-server, then an exception condition is raised: *feature not supported — multiple server transactions*.

2) Case:

   a) If DEFAULT is specified and there is no default SQL-connection that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist*.

   b) Otherwise, if <connection name> does not identify an SQL-session that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist*.

3) If the SQL-connection identified by <connection object> cannot be selected, then an exception condition is raised: *connection exception — connection failure*.

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

   NOTE 421 — The SQL-session context information is defined in Subclause 4.37, "SQL-sessions".

5) The SQL-connection identified by <connection object> becomes the current SQL-connection and the SQL-session associated with that SQL-connection becomes the current SQL-session. All SQL-session context information is restored to the same state as at the time the SQL-connection became dormant.

NOTE 422 — The SQL-session context information is defined in Subclause 4.37, "SQL-sessions".

6) The SQL-server for the subsequent execution of <externally-invoked procedure>s in any SQL-client modules associated with the SQL-agent are set to that of the current SQL-connection.

Conformance Rules

1) Without Feature F771, "Connection management", conforming SQL language shall not contain a <set connection statement>.

17.3 <disconnect statement>

Function

Terminate an SQL-connection.

Format

```
<disconnect statement> ::= DISCONNECT <disconnect object>

<disconnect object> ::=
    <connection object>
  | ALL
  | CURRENT
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) If <connection name> is specified and <connection name> does not identify an SQL-connection that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist.*

2) If DEFAULT is specified and there is no default SQL-connection that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist.*

3) If CURRENT is specified and there is no current SQL-connection for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist.*

4) Let *C* be the current SQL-connection.

5) Let *L* be a list of SQL-connections. If a <connection name> is specified, then *L* is that SQL-connection. If CURRENT is specified, then *L* is the current SQL-connection. If ALL is specified, then *L* is a list representing every SQL-connection that is current or dormant for the current SQL-agent, in an implementation-dependent order. If DEFAULT is specified, then *L* is the default SQL-connection.

6) If any SQL-connection in *L* is active, then an exception condition is raised: *invalid transaction state — active SQL-transaction.*

7) For every SQL-connection *C1* in *L*, treating the SQL-session *S1* identified by *C1* as the current SQL-session, all of the actions that are required after the last call of a <externally-invoked procedure> by an SQL-agent,

except for the execution of a <rollback statement> or a <commit statement>, are performed. *C1* is terminated, regardless of any exception condition that might occur during the disconnection process.

NOTE 423 — See the General Rules of Subclause 13.1, "<SQL-client module definition>", for the actions to be performed after the last call of a <externally-invoked procedure> by an SQL-agent.

8) If any error is detected during execution of a <disconnect statement>, then a completion condition is raised: *warning — disconnect error.*

9) If *C* is contained in *L*, then there is no current SQL-connection following the execution of the <disconnect statement>. Otherwise, *C* remains the current SQL-connection.

Conformance Rules

1) Without Feature F771, "Connection management", conforming SQL language shall not contain a <disconnect statement>.

This page intentionally left blank.

18 Session management

18.1 <set session characteristics statement>

Function

Set one or more characteristics for the current SQL-session.

Format

```
<set session characteristics statement> ::=
    SET SESSION CHARACTERISTICS AS <session characteristic list>

<session characteristic list> ::=
    <session characteristic> [ { <comma> <session characteristic> }... ]

<session characteristic> ::= <transaction characteristics>
```

Syntax Rules

1) None of <isolation level>, <transaction access mode>, and <diagnostics size> shall be specified more than once in a single <session characteristic list>.

Access Rules

None.

General Rules

1) For each <transaction characteristics> contained in the <session characteristic list>, the enduring transaction characteristics of the SQL-session are set to the values explicitly specified in the <transaction characteristics>; enduring characteristics corresponding to <transaction characteristics> values not explicitly specified are unchanged.

Conformance Rules

1) Without Feature F761, "Session management", conforming SQL language shall not contain a <set session characteristics statement>.

2) Without Feature F111, "Isolation levels other than SERIALIZABLE", conforming SQL language shall not contain a <set session characteristics statement> that contains a <level of isolation> other than SERIALIZABLE.

18.2 <set session user identifier statement>

Function

Set the SQL-session user identifier and the current user identifier of the current SQL-session context.

Format

```
<set session user identifier statement> ::=
    SET SESSION AUTHORIZATION <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) If a <set session user identifier statement> is executed and an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction.*

2) Let *S* be <value specification> and let *V* be the character string that is the value of

```
TRIM ( BOTH ' ' FROM S )
```

3) If *V* does not conform to the Format and Syntax Rules of an <authorization identifier>, then an exception condition is raised: *invalid authorization specification.*

4) Whether or not the SQL-session user identifier can be set to a different <user identifier> is implementation-defined, as are any restrictions pertaining to such changes.

5) If the current user identifier and the current role name are restricted from setting the user identifier to *V*, then an exception condition is raised: *invalid authorization specification.*

6) The SQL-session user identifier of the current SQL-session context is set to *V*.

7) The current user identifier is set to *V*.

8) The SQL-session role name and the current role name are removed.

Conformance Rules

1) Without Feature F321, "User authorization", conforming SQL language shall not contain a <set session user identifier statement>.

18.3 <set role statement>

Function

Set the current role name for the current SQL-session context.

Format

```
<set role statement> ::= SET ROLE <role specification>

<role specification> ::=
    <value specification>
  | NONE
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) If a <set role statement> is executed and an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction.*

2) Let *S* be <value specification> and let *V* be the character string that is the value of

```
TRIM ( BOTH ' ' FROM S )
```

3) If *V* does not conform to the Format and Syntax Rules of a <role name>, then an exception condition is raised: *invalid role specification.*

4) If no role authorization descriptor exists that indicates that the role identified by *V* has been granted to either the current user identifier or to PUBLIC, then an exception condition is raised: *invalid role specification.*

5) Case:

   a) If NONE is specified, then

      Case:

      i) If there is no current user identifier, then an exception condition is raised: *invalid role specification.*

      ii) Otherwise, the current role name is removed.

b) Otherwise, the current role name is set to *V*.

Conformance Rules

1) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <set role statement>.

18.4 <set local time zone statement>

Function

Set the current default time zone displacement for the current SQL-session.

Format

```
<set local time zone statement> ::= SET TIME ZONE <set time zone value>

<set time zone value> ::=
    <interval value expression>
  | LOCAL
```

Syntax Rules

1) The declared type of the <interval value expression> immediately contained in the <set time zone value> shall be INTERVAL HOUR TO MINUTE.

Access Rules

None.

General Rules

1) Case:

   a) If LOCAL is specified, then the current default time zone displacement of the current SQL-session is set to the original time zone displacement of the current SQL-session.

   b) Otherwise,

      Case:

      i) If the value of the <interval value expression> is not the null value and is between INTERVAL –'12:59' and INTERVAL +'14:00', then the current default time zone displacement of the current SQL-session is set to the value of the <interval value expression>.

      ii) Otherwise, an exception condition is raised: *data exception — invalid time zone displacement value.*

Conformance Rules

1) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <set local time zone statement>.

18.5 <set catalog statement>

Function

Set the default catalog name for unqualified <schema name>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s that are invoked directly.

Format

```
<set catalog statement> ::= SET <catalog name characteristic>

<catalog name characteristic> ::= CATALOG <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

```
TRIM ( BOTH ' ' FROM S )
```

2) If *V* does not conform to the Format and Syntax Rules of a <catalog name>, then an exception condition is raised: *invalid catalog name.*

3) The default catalog name of the current SQL-session is set to *V*.

Conformance Rules

1) Without Feature F651, "Catalog name qualifiers", conforming SQL language shall not contain a <set catalog statement>.

2) Without Feature F761, "Session management", conforming SQL language shall not contain a <set catalog statement>.

18.6 <set schema statement>

Function

Set the default schema name for unqualified <schema qualified name>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s that are invoked directly.

Format

```
<set schema statement> ::= SET <schema name characteristic>

<schema name characteristic> ::= SCHEMA <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

2) If *V* does not conform to the Format and Syntax Rules of a <schema name>, then an exception condition is raised: *invalid schema name.*

3) Case:

   a) If *V* conforms to the Format and Syntax Rules for a <schema name> that contains a <catalog name>, then let *X* be the <catalog name> part and let *Y* be the <unqualified schema name> part of *V*. The following statement is implicitly executed:

      ```
      SET CATALOG 'X'
      ```

      and the <set schema statement> is effectively replaced by:

      ```
      SET SCHEMA 'Y'
      ```

   b) Otherwise, the default unqualified schema name of the current SQL-session is set to *V*.

Conformance Rules

1) Without Feature F761, "Session management", conforming SQL language shall not contain a <set schema statement>.

18.7 <set names statement>

Function

Set the default character set name for <character string literal>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s that are invoked directly.

Format

```
<set names statement> ::= SET <character set name characteristic>

<character set name characteristic> ::= NAMES <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

```
TRIM ( BOTH ' ' FROM S )
```

2) If *V* does not conform to the Format and Syntax Rules of a <character set name>, then an exception condition is raised: *invalid character set name.*

3) The default character set name of the current SQL-session is set to *V*.

Conformance Rules

1) Without and Feature F461, "Named character sets", conforming SQL language shall not contain a <set names statement>.

2) Without Feature F761, "Session management", conforming SQL language shall not contain a <set names statement>.

18.8 <set path statement>

Function

Set the SQL-path used to determine the subject routine of <routine invocation>s with unqualified <routine name>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s, respectively, that are invoked directly. The SQL-path remains the current SQL-path of the SQL-session until another SQL-path is successfully set.

Format

```
<set path statement> ::= SET <SQL-path characteristic>

<SQL-path characteristic> ::= PATH <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

```
TRIM ( BOTH ' ' FROM S )
```

   a) If *V* does not conform to the Format and Syntax Rules of a <schema name list>, then an exception condition is raised: *invalid schema name list specification*.

   b) The SQL-path of the current SQL-session is set to *V*.

NOTE 424 — A <set path statement> that is executed between a <prepare statement> and an <execute statement> has no effect on the prepared statement.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", Conforming SQL language shall not contain a <set path statement>.

18.9 <set transform group statement>

Function

Set the group name that identifies the group of transform functions for mapping values of user-defined types to predefined data types.

Format

```
<set transform group statement> ::= SET <transform group characteristic>

<transform group characteristic> ::=
    DEFAULT TRANSFORM GROUP <value specification>
  | TRANSFORM GROUP FOR TYPE <path-resolved user-defined type name> <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

2) If <path-resolved user-defined type name> is specified, then let *UDT* be the user-defined type identified by that <path-resolved user-defined type name>.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

   a) If *V* does not conform to the Format and Syntax Rules of a <group name>, then an exception condition is raised: *invalid transform group name specification*.

   b) Case:

      i) If <path-resolved user-defined type name> is specified, then the transform group name corresponding to all subtypes of *UDT* for the current SQL-session is set to *V*.

      ii) Otherwise, the default transform group name for the current SQL-session is set to *V*.

NOTE 425 — A <set transform group statement> that is executed after a <prepare statement> has no effect on the prepared statement.

Conformance Rules

1) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <set transform group statement>.

18.10 <set session collation statement>

Function

Set the SQL-session collation of the SQL-session for one or more character sets. An SQL-session collation remains effective until another SQL-session collation for the same character set is successfully set.

Format

```
<set session collation statement> ::=
    SET COLLATION <collation specification> [ FOR <character set specification list> ]
  | SET NO COLLATION [ FOR <character set specification list> ]

<collation specification> ::= <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of
`TRIM ( BOTH ' ' FROM S )`

   a) If *V* does not conform to the Format and Syntax Rules of a <collation name>, then an exception condition is raised: *invalid collation name.*

   b) Let *CO* be the collation identified by the <collation name> contained in *V*.

      Case:

      i) If <character set specification list> is specified, then

         Case:

         1) If the collation specified by *CO* is not applicable to any character set identified by a <character set specification>, then an exception condition is raised: *invalid collation name.*

         2) Otherwise, for each character set specified, the SQL-session collation for that character set in the current SQL-session is set to *CO*.

      ii) Otherwise, the character sets for which the SQL-session collations are set to *CO* are implementation-defined.

2) If SET NO COLLATION is specified, then

Case:

a) If <character set specification list> is specified, then, for each character set specified, the SQL-session collation for that character set in the current SQL-session is set to *none*.

b) Otherwise, the SQL-session collation for every character set in the current SQL-session is set to *none*.

Conformance Rules

1) Without Feature F693, "SQL-session and client module collations", conforming SQL language shall not contain a <set session collation statement>.

This page intentionally left blank.

19 Dynamic SQL

19.1 Description of SQL descriptor areas

Function

Specify the identifiers, data types, and codes used in SQL item descriptor areas.

Syntax Rules

1) An SQL item descriptor area comprises the items specified in Table 24, "Data types of <key word>s used in SQL item descriptor areas".

2) An SQL descriptor area comprises the items specified in Table 23, "Data types of <key word>s used in the header of SQL descriptor areas", and one or more occurrences of an SQL item descriptor area.

3) Given an SQL item descriptor area *IDA* in which the value of LEVEL is N, the *immediately subordinate descriptor areas* of *IDA* are those SQL item descriptor areas in which the value of LEVEL is *N*+1 and whose position in the SQL descriptor area follows that of *IDA* and precedes that of any SQL item descriptor area in which the value of LEVEL is less than *N*+1.

   The *subordinate descriptor areas* of *IDA* are those SQL item descriptor areas that are immediately subordinate descriptor areas of *IDA* or that are subordinate descriptor areas of an SQL item descriptor area that is immediately subordinate to *IDA*.

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

   Case:

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

   b) If *DT* is a collection type, then the descriptor of the associated element type of *DT*.

   The *subordinate descriptors* of *DE* are those descriptors that are immediately subordinate descriptors of *DE* or that are subordinate descriptors of a descriptor that is immediately subordinate to *DE*.

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

   a) SDE_1 is in the first ordinal position.

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

6) An item descriptor area *IDA* is *valid* if and only if TYPE indicates a code defined in Table 25, "Codes used for SQL data types in Dynamic SQL", and one of the following is true:

Case:

a) TYPE indicates CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, LENGTH is a valid length value for TYPE, and CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the fully qualified name of a character set that is valid for TYPE.

b) TYPE indicates CHARACTER LARGE OBJECT LOCATOR.

c) TYPE indicates BINARY LARGE OBJECT and LENGTH is a valid length value for the BINARY LARGE OBJECT data type.

d) TYPE indicates BINARY LARGE OBJECT LOCATOR.

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

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

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

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

i) TYPE indicates BOOLEAN.

j) TYPE indicates a <datetime type>, DATETIME_INTERVAL_CODE is a code specified in Table 26, "Codes associated with datetime data types in Dynamic SQL", and PRECISION is a valid value for the <time precision> or <timestamp precision> of the indicated datetime data type.

k) TYPE indicates an <interval type>, DATETIME_INTERVAL_CODE is a code specified in Table 27, "Codes used for <interval qualifier>s in Dynamic SQL", and DATETIME_INTERVAL_PRECISION and PRECISION are valid values for <interval leading field precision> and <interval fractional seconds precision> for an <interval qualifier>.

l) TYPE indicates USER-DEFINED TYPE LOCATOR and USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are the fully qualified name of a valid user-defined type.

m) TYPE indicates REF, LENGTH is the length in octets for the REF type, and USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are a valid qualified user-defined type name, and SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME are a valid qualified table name.

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

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

p) TYPE indicates MULTISET or MULTISET LOCATOR, there is exactly one immediately subordinate descriptor area of *IDA*, and that SQL item descriptor area is valid.

q) TYPE indicates an implementation-defined data type.

7) The declared type *T* of a <simple value specification> or a <simple target specification> *SVT* is said to *match* the data type specified by a valid item descriptor area *IDA* if and only if one of the following conditions is true.

Case:

a) TYPE indicates CHARACTER and *T* is specified by CHARACTER(*L*), where *L* is the value of LENGTH and the <character set specification> formed by the values of CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME identifies the character set of *SVT*.

b) Either TYPE indicates CHARACTER VARYING and *T* is specified by CHARACTER VARYING(*L*) or type indicates CHARACTER LARGE OBJECT and *T* is specified by CHARACTER LARGE OBJECT(*L*), where the <character set specification> formed by the values of CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME identifies the character set of *SVT* and

Case:

i) *SVT* is a <simple value specification> and *L* is the value of LENGTH.

ii) *SVT* is a <simple target specification> and *L* is not less than the value of LENGTH.

c) TYPE indicates CHARACTER LARGE OBJECT LOCATOR and *T* is specified by CHARACTER LARGE OBJECT LOCATOR.

d) TYPE indicates BINARY LARGE OBJECT and *T* is specified by BINARY LARGE OBJECT(*L*) and

Case:

i) *STV* is a <simple value specification> and *L* is the value of LENGTH.

ii) *STV* is a <simple target specification> and *L* is not less than the value of LENGTH.

e) TYPE indicates BINARY LARGE OBJECT LOCATOR and *T* is specified by BINARY LARGE OBJECT LOCATOR.

f) TYPE indicates NUMERIC and *T* is specified by NUMERIC(*P*,*S*), where *P* is the value of PRECISION and *S* is the value of SCALE.

g) TYPE indicates DECIMAL and *T* is specified by DECIMAL(*P*,*S*), where *P* is the value of PRECISION and *S* is the value of SCALE.

h) TYPE indicates SMALLINT and *T* is specified by SMALLINT.

i) TYPE indicates INTEGER and *T* is specified by INTEGER.

j) TYPE indicates BIGINT and *T* is specified by BIGINT.

k) TYPE indicates FLOAT and *T* is specified by FLOAT(*P*), where *P* is the value of PRECISION.

l) TYPE indicates REAL and *T* is specified by REAL.

m) TYPE indicates DOUBLE PRECISION and *T* is specified by DOUBLE PRECISION.

n) TYPE indicates BOOLEAN and *T* is specified by BOOLEAN.

o) TYPE indicates USER-DEFINED TYPE and *T* is specified by USER-DEFINED TYPE LOCATOR, where the values of USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are the fully qualified name of the associated user-defined type of *SVT*.

p) TYPE indicates REF and *T* is specified by REF, where the <user-defined type name> formed by the values of USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME identifies the row type of *SVT*, and SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME identify the scope of the reference type.

q) TYPE indicates ROW, and *T* is a row type with degree *D*, where *D* is the value of DEGREE, and the data type of the *i*-th field of *SVT* matches the data type specified by the *i*-th immediately subordinate descriptor area of *IDA*.

r) TYPE indicates ARRAY and *T* is an array type with maximum cardinality *C* and the data type of the element type of *T* matches the immediately subordinate descriptor area of *IDA*, and

Case:

i) *SVT* is a <simple value specification> and *C* is the value of CARDINALITY.

ii) *SVT* is a <simple target specification> and *C* is not less than the value of CARDINALITY.

s) TYPE indicates ARRAY LOCATOR and *T* is an array locator type whose associated array type has maximum cardinality *C* and the data type of the element type of the associated array type of *T* matches the immediately subordinate descriptor area of *IDA*, and

Case:

i) *SVT* is a <simple value specification> and *C* is the value of CARDINALITY.

ii) *SVT* is a <simple target specification> and *C* is not less than the value of CARDINALITY.

t) TYPE indicates MULTISET and *T* is a multiset type and the data type of the element type of *T* matches the immediately subordinate descriptor area of *IDA*.

u) TYPE indicates MULTISET LOCATOR and *T* is a multiset locator type and the data type of the element type of *T* matches the immediately subordinate descriptor area of *IDA*.

v) TYPE indicates a data type from Table 25, "Codes used for SQL data types in Dynamic SQL", other than an implementation-defined data type and *T* satisfies the implementation-defined rules for matching that data type.

w) TYPE indicates an implementation-defined data type and *T* satisfies the implementation-defined rules for matching that data type.

8) A data type *DT* is said to be *represented* by an SQL item descriptor area if a <simple value specification> of type *DT* matches the SQL item descriptor area.

Table 23 — Data types of <key word>s used in the header of SQL descriptor areas

<key word>	Data Type
COUNT	exact numeric with scale 0 (zero)
DYNAMIC_FUNCTION	character string with character set SQL_IDENTIFIER and length not less than 128 characters
DYNAMIC_FUNCTION_CODE	exact numeric with scale 0 (zero)
KEY_TYPE	exact numeric with scale 0 (zero)
TOP_LEVEL_COUNT	exact numeric with scale 0 (zero)

Table 24 — Data types of <key word>s used in SQL item descriptor areas

<key word>	Data Type
CARDINALITY	exact numeric with scale 0 (zero)
CHARACTER_SET_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
CHARACTER_SET_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
CHARACTER_SET_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
COLLATION_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
COLLATION_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
COLLATION_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
DATA	matches the data type represented by the SQL item descriptor area
DATETIME_INTERVAL_CODE	exact numeric with scale 0 (zero)
DATETIME_INTERVAL_PRECISION	exact numeric with scale 0 (zero)

<key word>	Data Type
DEGREE	exact numeric with scale 0 (zero)
INDICATOR	exact numeric with scale 0 (zero)
KEY_MEMBER	exact numeric with scale 0 (zero)
LENGTH	exact numeric with scale 0 (zero)
LEVEL	exact numeric with scale 0 (zero)
NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
NULLABLE	exact numeric with scale 0 (zero)
OCTET_LENGTH	exact numeric with scale 0 (zero)
PARAMETER_MODE	exact numeric with scale 0 (zero)
PARAMETER_ORDINAL_POSITION	exact numeric with scale 0 (zero)
PARAMETER_SPECIFIC_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
PARAMETER_SPECIFIC_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
PARAMETER_SPECIFIC_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
PRECISION	exact numeric with scale 0 (zero)
RETURNED_CARDINALITY	exact numeric with scale 0 (zero)
RETURNED_LENGTH	exact numeric with scale 0 (zero)
RETURNED_OCTET_LENGTH	exact numeric with scale 0 (zero)
SCALE	exact numeric with scale 0 (zero)
SCOPE_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
SCOPE_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
SCOPE_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters

<key word>	Data Type
TYPE	exact numeric with scale 0 (zero)
UNNAMED	exact numeric with scale 0 (zero)
USER_DEFINED_TYPE_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
USER_DEFINED_TYPE_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
USER_DEFINED_TYPE_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
USER_DEFINED_TYPE_CODE	exact numeric with scale 0 (zero)

NOTE 426 — "Matches" and "represented by", as applied to the relationship between a data type and an SQL item descriptor area are defined in the Syntax Rules of this Subclause.

Access Rules

None.

General Rules

1) Table 25, "Codes used for SQL data types in Dynamic SQL", specifies the codes associated with the SQL data types.

Table 25 — Codes used for SQL data types in Dynamic SQL

Data Type	Code
Implementation-defined data types	< 0 (zero)
ARRAY	50
ARRAY LOCATOR	51
BIGINT	25
BLOB	30
BLOB LOCATOR	31
BOOLEAN	16
CHARACTER	1 (one)

Data Type	Code
CHARACTER VARYING	12
CLOB	40
CLOB LOCATOR	41
DATE, TIME WITHOUT TIME ZONE, TIME WITH TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
INTERVAL	10
MULTISET	55
MULTISET LOCATOR	56
NUMERIC	2
REAL	7
SMALLINT	5
USER-DEFINED TYPE LOCATOR	18
ROW TYPE	19
REF	20
User-defined types	17

2) Table 26, "Codes associated with datetime data types in Dynamic SQL", specifies the codes associated with the datetime data types.

Table 26 — Codes associated with datetime data types in Dynamic SQL

Datetime Data Type	Code
DATE	1 (one)

Datetime Data Type	Code
TIME WITH TIME ZONE	4
TIME WITHOUT TIME ZONE	2
TIMESTAMP WITH TIME ZONE	5
TIMESTAMP WITHOUT TIME ZONE	3

3) Table 27, "Codes used for <interval qualifier>s in Dynamic SQL", specifies the codes associated with <interval qualifier>s for interval data types.

Table 27 — Codes used for <interval qualifier>s in Dynamic SQL

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

4) The value of DYNAMIC_FUNCTION is a character string that identifies the type of the prepared or executed SQL-statement. Table 31, "SQL-statement codes", specifies the identifier of the SQL-statements.

5) The value of DYNAMIC_FUNCTION_CODE is a number that identifies the type of the prepared or executed SQL-statement. Table 31, "SQL-statement codes", specifies the identifier of the SQL-statements.

6) Table 28, "Codes used for input/output SQL parameter modes in Dynamic SQL", specifies the codes used for the PARAMETER_MODE item descriptor field when describing a <call statement>.

Table 28 — Codes used for input/output SQL parameter modes in Dynamic SQL

Parameter mode	Code
PARAMETER_MODE_IN	1 (one)
PARAMETER_MODE_INOUT	2
PARAMETER_MODE_OUT	4

7) Table 29, "Codes associated with user-defined types in Dynamic SQL", specifies the codes associated with user-defined types.

Table 29 — Codes associated with user-defined types in Dynamic SQL

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

Conformance Rules

None.

19.2 <allocate descriptor statement>

Function

Allocate an SQL descriptor area.

Format

```
<allocate descriptor statement> ::=
    ALLOCATE [ SQL ] DESCRIPTOR <descriptor name> [ WITH MAX <occurrences> ]

<occurrences> ::= <simple value specification>
```

Syntax Rules

1) The declared type of <occurrences> shall be exact numeric with scale 0 (zero).

2) If WITH MAX <occurrences> is not specified, then an implementation-defined default value for <occurrences> that is greater than 0 (zero) is implicit.

Access Rules

None.

General Rules

1) Let *S* be the <simple value specification> that is immediately contained in <descriptor name> and let *V* be the character string that is the result of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

   If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid SQL descriptor name.*

2) Case:

   a) If an SQL descriptor area whose name is *V* and whose scope is specified by the <scope option> immediately contained in a <descriptor name> is currently allocated, then an exception condition is raised: *invalid SQL descriptor name.*

   b) Otherwise, the <allocate descriptor statement> allocates an SQL descriptor area whose name is *V* and whose scope is specified by the <scope option> immediately contained in a <descriptor name>. The SQL descriptor area will have at least <occurrences> number of SQL item descriptor areas. The value of LEVEL in each of the item descriptor areas is set to 0 (zero). The values of all other fields in the SQL descriptor area are initially undefined.

3) If <occurrences> is less than 1 (one) or is greater than an implementation-defined maximum value, then an exception condition is raised: *dynamic SQL error — invalid descriptor index.* The maximum number of SQL descriptor areas that can be allocated at one time is implementation-defined.

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain an <occurrences> that is not a <literal>.

2) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <allocate descriptor statement>.

19.3 <deallocate descriptor statement>

Function

Deallocate an SQL descriptor area.

Format

```
<deallocate descriptor statement> ::=
    DEALLOCATE [ SQL ] DESCRIPTOR <descriptor name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Case:

   a) If an SQL descriptor area is not currently allocated whose name is the value of the <simple value specification> immediately contained in <descriptor name> and whose scope is specified by the <scope option> immediately contained in <descriptor name>, then an exception condition is raised: *invalid SQL descriptor name.*

   b) Otherwise, the <deallocate descriptor statement> deallocates an SQL descriptor area whose name is the value of the <simple value specification> immediately contained in <descriptor name> and whose scope is specified by the <scope option> immediately contained in <descriptor name>.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <deallocate descriptor statement>.

19.4 <get descriptor statement>

Function

Get information from an SQL descriptor area.

Format

```
<get descriptor statement> ::=
    GET [ SQL ] DESCRIPTOR <descriptor name> <get descriptor information>

<get descriptor information> ::=
    <get header information> [ { <comma> <get header information> }... ]
  | VALUE <item number> <get item information>
    [ { <comma> <get item information> }... ]

<get header information> ::=
    <simple target specification 1> <equals operator> <header item name>

<header item name> ::=
    COUNT
  | KEY_TYPE
  | DYNAMIC_FUNCTION
  | DYNAMIC_FUNCTION_CODE
  | TOP_LEVEL_COUNT

<get item information> ::=
    <simple target specification 2> <equals operator> <descriptor item name>

<item number> ::= <simple value specification>

<simple target specification 1> ::= <simple target specification>

<simple target specification 2> ::= <simple target specification>

<descriptor item name> ::=
    CARDINALITY
  | CHARACTER_SET_CATALOG
  | CHARACTER_SET_NAME
  | CHARACTER_SET_SCHEMA
  | COLLATION_CATALOG
  | COLLATION_NAME
  | COLLATION_SCHEMA
  | DATA
  | DATETIME_INTERVAL_CODE
  | DATETIME_INTERVAL_PRECISION
  | DEGREE
  | INDICATOR
  | KEY_MEMBER
  | LENGTH
  | LEVEL
  | NAME
  | NULLABLE
  | OCTET_LENGTH
```

```
| PARAMETER_MODE
| PARAMETER_ORDINAL_POSITION
| PARAMETER_SPECIFIC_CATALOG
| PARAMETER_SPECIFIC_NAME
| PARAMETER_SPECIFIC_SCHEMA
| PRECISION
| RETURNED_CARDINALITY
| RETURNED_LENGTH
| RETURNED_OCTET_LENGTH
| SCALE
| SCOPE_CATALOG
| SCOPE_NAME
| SCOPE_SCHEMA
| TYPE
| UNNAMED
| USER_DEFINED_TYPE_CATALOG
| USER_DEFINED_TYPE_NAME
| USER_DEFINED_TYPE_SCHEMA
| USER_DEFINED_TYPE_CODE
```

Syntax Rules

1) The declared type of <item number> shall be exact numeric with scale 0 (zero).

2) For each <get header information>, the declared type of <simple target specification 1> shall be that shown in the Data Type column of the row in Table 23, "Data types of <key word>s used in the header of SQL descriptor areas", whose <key word> column value is equivalent to <header item name>.

3) For each <get item information>, the declared type of <simple target specification 2> shall be that shown in the Data Type column of the row in Table 24, "Data types of <key word>s used in SQL item descriptor areas", whose <key word> column value is equivalent to <descriptor item name>.

Access Rules

None.

General Rules

1) If <descriptor name> identifies an SQL descriptor area that is not currently allocated whose name is the value of the <simple value specification> immediately contained in <descriptor name> and whose scope is specified by the <scope option> immediately contained in <descriptor name>, then an exception condition is raised: *invalid SQL descriptor name.*

2) If the <item number> specified in a <get descriptor statement> is greater than the value of <occurrences> specified when the <descriptor name> was allocated or less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index.*

3) If the <item number> specified in a <get descriptor statement> is greater than the value of COUNT, then a completion condition is raised: *no data.*

4) If the declared type of the <simple target specification> associated with the keyword DATA does not match the data type represented by the item descriptor area, then an exception condition is raised: *data exception — error in assignment.*

NOTE 427 — "Match" and "represented by" are defined in the Syntax Rules of Subclause 19.1, "Description of SQL descriptor areas".

5) Let *i* be the value of the <item number> contained in <get descriptor information>. Let *IDA* be the *i*-th item descriptor area. If a <get item information> specifies DATA, then:

a) If *IDA* is subordinate to an item descriptor area whose TYPE field indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then an exception condition is raised: *dynamic SQL error — undefined DATA value.*

b) If the value of TYPE in *IDA* indicates ROW, then an exception condition is raised: *dynamic SQL error — undefined DATA value.*

c) If the value of INDICATOR is negative and no <get item information> specifies INDICATOR, then an exception condition is raised: *data exception — null value, no indicator parameter.*

6) If an exception condition is raised in a <get descriptor statement>, then the values of all targets specified by <simple target specification 1> and <simple target specification 2> are implementation-dependent.

7) A <get descriptor statement> retrieves values from the SQL descriptor area and item specified by <descriptor name>. For each item, the value that is retrieved is the one established by the most recently executed <allocate descriptor statement>, <set descriptor statement>, or <describe statement> that references the specified SQL descriptor area and item. The value retrieved by a <get descriptor statement> for any field whose value is undefined is implementation-dependent.

Case:

a) If <get descriptor information> contains one or more <get header information>s, then for each <get header information> specified, the value of <simple target specification 1> is set to the value *V* in the SQL descriptor area of the field identified by the <header item name> by applying the General Rules of Subclause 9.2, "Store assignment", to <simple target specification 1> and *V* as *TARGET* and *VALUE*, respectively.

b) If <get descriptor information> contains one or more <get item information>s, then:

i) Let *i* be the value of the <item number> contained in the <get descriptor information>.

ii) For each <get item information> specified, the value of <simple target specification 2> is set to the value *V* in the *i*-th SQL item descriptor area of the field identified by the <descriptor item name> by applying the General Rules of Subclause 9.2, "Store assignment", to <simple target specification 2> and *V* as *TARGET* and *VALUE*, respectively.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <get descriptor statement>.

2) Without Feature T301, "Functional dependencies", conforming SQL language shall not contain a <descriptor item name> that contains KEY_MEMBER.

19.5 <set descriptor statement>

Function

Set information in an SQL descriptor area.

Format

```
<set descriptor statement> ::=
    SET [ SQL ] DESCRIPTOR <descriptor name> <set descriptor information>

<set descriptor information> ::=
    <set header information> [ { <comma> <set header information> }... ]
  | VALUE <item number> <set item information>
    [ { <comma> <set item information> }... ]

<set header information> ::=
    <header item name> <equals operator> <simple value specification 1>

<set item information> ::=
    <descriptor item name> <equals operator> <simple value specification 2>

<simple value specification 1> ::= <simple value specification>

<simple value specification 2> ::= <simple value specification>
```

Syntax Rules

1) For each <set header information>, <header item name> shall not be KEY_TYPE, TOP_LEVEL_COUNT, DYNAMIC_FUNCTION, or DYNAMIC_FUNCTION_CODE, and the declared type of <simple value specification 1> shall be that in the Data Type column of the row of Table 23, "Data types of <key word>s used in the header of SQL descriptor areas", whose <key word> column value is equivalent to <header item name>.

2) For each <set item information>, the value of <descriptor item name> shall not be RETURNED_LENGTH, RETURNED_OCTET_LENGTH, RETURNED_CARDINALITY, OCTET_LENGTH, NULLABLE, KEY_MEMBER, COLLATION_CATALOG, COLLATION_SCHEMA, COLLATION_NAME, NAME, UNNAMED, PARAMETER_MODE, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, PARAMETER_SPECIFIC_NAME, or USER_DEFINED_TYPE_CODE. Other alternatives for <descriptor item name> shall not be specified more than once in a <set descriptor statement>. The declared type of <simple value specification 2> shall be that shown in the Data Type column of the row in Table 24, "Data types of <key word>s used in SQL item descriptor areas", whose <key word> column value is equivalent to <descriptor item name>.

3) If the <descriptor item name> specifies DATA, then <simple value specification 2> shall not be a <literal>.

Access Rules

None.

General Rules

1) If <descriptor name> identifies an SQL descriptor area that is not currently allocated whose name is the value of the <simple value specification> immediately contained in <descriptor name> and whose scope is specified by the <scope option> immediately contained in <descriptor name>, then an exception condition is raised: *invalid SQL descriptor name.*

2) If the <item number> specified in a <set descriptor statement> is greater than the value of <occurrences> specified when the <descriptor name> was allocated or less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index.*

3) When more than one value is set in a single <set descriptor statement>, the values are effectively assigned in the following order: LEVEL, TYPE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, PRECISION, SCALE, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, SCOPE_NAME, LENGTH, INDICATOR, DEGREE, CARDINALITY, and DATA.

   When any value other than DATA is set, the value of DATA becomes undefined.

4) For every <set item information> specified, let *DIN* be the <descriptor item name>, let *V* be the value of the <simple value specification 2>, let *N* be the value of <item number>, and let *IDA* be the *N*-th item descriptor area.

   Case:

   a) If *DIN* is DATA, then:

      i) If *IDA* is subordinate to an item descriptor area whose TYPE field indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then an exception condition is raised: *dynamic SQL error — invalid DATA target.*

      ii) If TYPE in *IDA* indicates ROW, then an exception condition is raised: *dynamic SQL error — invalid DATA target.*

      iii) If the most specific type of *V* does not match the data type specified by the item descriptor area, then an exception condition is raised: *data exception — error in assignment.*

      NOTE 428 — "Match" is defined in the Syntax Rules of Subclause 19.1, "Description of SQL descriptor areas".

      iv) The value of DATA in *IDA* is set to *V*.

   b) If *DIN* is LEVEL, then:

      i) If *N* is 1 (one) and *V* is not 0 (zero), then an exception condition is raised: *dynamic SQL error — invalid LEVEL value.*

      ii) If *N* is greater than 1 (one), then let *PIDA* be *IDA*'s immediately preceding item descriptor area and let *K* be its LEVEL value.

         1) If *V* = *K*+1 and TYPE in *PIDA* does not indicate ROW, ARRAY, ARRAY LOCATOR, MULTISET, MULTISET LOCATOR, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value.*

2) If $V > K+1$, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value.*

3) If $V < K+1$, then let $OIDA_i$ be the i-th item descriptor area to which *PIDA* is subordinate and whose TYPE field indicates ROW, let NS_i be the number of immediately subordinate descriptor areas of $OIDA_i$ between $OIDA_i$ and *IDA* and let D_i be the value of DEGREE in $OIDA_i$.

A) For each $OIDA_i$ whose LEVEL value is greater than V, if D_i is not equal to NS_i, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value.*

B) If K is not 0 (zero), then let $OIDA_j$ be the $OIDA_i$ whose LEVEL value is K. If there exists no such $OIDA_j$ or D_j is not greater than NS_j, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value.*

iii) The value of LEVEL in *IDA* is set to V.

c) If *DIN* is TYPE, then:

i) The value of TYPE in *IDA* is set to V.

ii) The value of all fields other than TYPE and LEVEL in *IDA* are set to implementation-dependent values.

iii) Case:

1) If V indicates CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA and CHARACTER_SET_NAME in *IDA* are set to the values for the default character set name for the SQL-session and LENGTH in *IDA* is set to 1 (one).

2) If V indicates CHARACTER LARGE OBJECT LOCATOR, then LENGTH in *IDA* is set to 1 (one).

3) If V indicates BINARY LARGE OBJECT, then LENGTH in *IDA* is set to 1 (one).

4) If V indicates BINARY LARGE OBJECT LOCATOR, then LENGTH in *IDA* is set to 1 (one).

5) If V indicates DATETIME, then PRECISION in *IDA* is set to 0 (zero).

6) If V indicates INTERVAL, then DATETIME_INTERVAL_PRECISION in *IDA* is set to 2.

7) If V indicates NUMERIC or DECIMAL, then SCALE in *IDA* is set to 0 (zero) and PRECISION in *IDA* is set to the implementation-defined default value for the precision of NUMERIC or DECIMAL data types, respectively.

8) If V indicates FLOAT, then PRECISION in *IDA* is set to the implementation-defined default value for the precision of the FLOAT data type.

d) If *DIN* is DATETIME_INTERVAL_CODE, then

Case:

i) If TYPE in *IDA* indicates DATETIME, then

Case:

1) If *V* indicates DATE, TIME, or TIME WITH TIME ZONE, then PRECISION in *IDA* is set to 0 (zero) and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

2) If *V* indicates TIMESTAMP or TIMESTAMP WITH TIME ZONE, then PRECISION in *IDA* is set to 6 and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

3) Otherwise, an exception condition is raised: *dynamic SQL error — invalid DATETIME_INTERVAL_CODE.*

ii) If TYPE in *IDA* indicates INTERVAL, then

Case:

1) If *V* indicates DAY TO SECOND, HOUR TO SECOND, MINUTE TO SECOND, or SECOND, then PRECISION in *IDA* is set to 6, DATETIME_INTERVAL_PRECISION in *IDA* is set to 2 and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

2) If *V* indicates YEAR, MONTH, DAY, HOUR, MINUTE, YEAR TO MONTH, DAY TO HOUR, DAY TO MINUTE, or HOUR TO MINUTE, then PRECISION in *IDA* is set to 0 (zero), DATETIME_INTERVAL_PRECISION in *IDA* is set to 2 and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

3) Otherwise, an exception condition is raised: *dynamic SQL error — invalid DATETIME_INTERVAL_CODE.*

iii) Otherwise, an exception condition is raised: *dynamic SQL error — invalid DATETIME_INTERVAL_CODE.*

e) Otherwise, the value of *DIN* in *IDA* is set to *V* by applying the General Rules of Subclause 9.2, "Store assignment", to the field of *IDA* identified by *DIN* and *V* as *TARGET* and *VALUE*, respectively. .

5) For each <set header information> specified, the value of the field identified by <header item name> is set to the value *V* of <simple value specification 1> by applying the General Rules of Subclause 9.2, "Store assignment", to the field identified by the <header item name> and *V* as *TARGET* and *VALUE*, respectively.

6) If an exception condition is raised in a <set descriptor statement>, then the values of all elements of the item descriptor area specified in the <set descriptor statement> are implementation-dependent.

7) Restrictions on changing TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME values resulting from the execution of a <describe statement> before execution of an <execute statement>, <dynamic open statement>, or <dynamic fetch statement> are implementation-defined.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <set descriptor statement>.

19.6 <prepare statement>

Function

Prepare a statement for execution.

Format

```
<prepare statement> ::=
    PREPARE <SQL statement name> [ <attributes specification> ]
    FROM <SQL statement variable>

<attributes specification> ::= ATTRIBUTES <attributes variable>

<attributes variable> ::= <simple value specification>

<SQL statement variable> ::= <simple value specification>

<preparable statement> ::=
    <preparable SQL data statement>
  | <preparable SQL schema statement>
  | <preparable SQL transaction statement>
  | <preparable SQL control statement>
  | <preparable SQL session statement>
  | <preparable implementation-defined statement>

<preparable SQL data statement> ::=
    <delete statement: searched>
  | <dynamic single row select statement>
  | <insert statement>
  | <dynamic select statement>
  | <update statement: searched>
  | <merge statement>
  | <preparable dynamic delete statement: positioned>
  | <preparable dynamic update statement: positioned>
  | <hold locator statement>
  | <free locator statement>

<preparable SQL schema statement> ::= <SQL schema statement>

<preparable SQL transaction statement> ::= <SQL transaction statement>

<preparable SQL control statement> ::= <SQL control statement>

<preparable SQL session statement> ::= <SQL session statement>

<dynamic select statement> ::= <cursor specification>

<preparable implementation-defined statement> ::= !! See the Syntax Rules.
```

Syntax Rules

1) The <simple value specification> of <SQL statement variable> shall not be a <literal>.

2) The declared types of each of <SQL statement variable> and <attributes variable> shall be character string.

3) The Format and Syntax Rules for <preparable implementation-defined statement> are implementation-defined.

4) A <preparable SQL control statement> shall not contain an <SQL procedure statement> that is not a <preparable statement>, nor shall it contain a <dynamic single row select statement> or a <dynamic select statement>.

Access Rules

None.

General Rules

1) Let *P* be the contents of the <SQL statement variable>. If *P* is an <SQL control statement>, then let *PS* be an <SQL procedure statement> contained in *P*.

2) Two subfields *SF1* and *SF2* of row types *RT1* and *RT2* are *corresponding subfields* if either *SF1* or *SF2* are positionally corresponding fields of *RT1* and *RT2*, respectively, or *SF1* and *SF2* are positionally corresponding fields of *RT1SF1* and *RT2SF2* and *RT1SF1* and *RT2SF2* are the declared types of corresponding subfields of *RT1* and *RT2* respectively.

3) If *P* does not conform to the Format, Syntax Rules, and Access Rules of a <preparable statement>, or if *P* contains a <simple comment> then an exception condition is raised: *syntax error or access rule violation.*

4) Let *DTGN* be the default transform group name and let *TFL* be the list of {user-defined type name — transform group name} pairs used to identify the group of transform functions for every user-defined type that is referenced in *P*. *DTGN* and *TFL* are not affected by the execution of a <set transform group statement> after *P* is prepared.

5) Let *DPV* be a <value expression> that is either a <dynamic parameter specification> or a <dynamic parameter specification> immediately contained in any number of <left paren> <right paren> pairs. Initially, the declared type of such a <value expression> is, by definition, undefined. A data type is *undefined* if it is neither a data type defined in this standard nor a data type defined by the implementation.

6) Let *MP* be the implementation-defined maximum value of <precision> for the NUMERIC data type. Let *ML* be the implementation-defined maximum value of <length> for the CHARACTER VARYING data type. For each <value expression> *DP* in *P* or *PS* that meets the criteria for *DPV* let *DT* denote its declared type. The syntactic substitutions specified in Subclause 14.12, "<set clause list>", shall not be applied until the data types of <dynamic parameter specification>s are determined by this General Rule.

   a) Case:

      i) If *DP* is immediately followed by an <interval qualifier> *IQ*, then *DT* is INTERVAL *IQ*.

      ii) If *DP* is the <numeric value expression> simply contained in an <array element reference>, then *DT* is NUMERIC (*MP*, 0).

      iii) If *DP* is the <string value expression> simply contained in a <char length expression> or an <octet length expression>, then *DT* is CHARACTER VARYING(*ML*) with an implementation-defined character set.

iv) If *DP* is either the <numeric value expression dividend> *X1* or the <numeric value expression divisor> *X2* simply contained in a <modulus expression>, then if *DP* is *X1* (*X2*), then *DT* is the declared type of *X2* (*X1*).

v) If *DP* is either *X1* or *X2* in a <position expression> of the form "POSITION <left paren> *X1* IN *X2* <right paren>", and if *DP* is *X1* (*X2*), then

Case:

1) If the declared type of *X2* (*X1*) is CHARACTER or CHARACTER VARYING with character set *CS*, then *DT* is CHARACTER VARYING (*ML*) with character set *CS*.

2) Otherwise, *DT* is the declared type of *X2* (*X1*).

vi) If *DP* is either *X2* or *X3* in a <string value function> of the form "SUBSTRING <left paren> *X1* FROM *X2* FOR *X3* <right paren>" or "SUBSTRING <left paren> *X1* FROM *X2* <right paren>", then *DT* is NUMERIC (*MP*, 0).

vii) If *DP* is either *X1*, *X2*, or *X3* in a <string value function> of the form "SUBSTRING (*X1* SIMILAR *X2* ESCAPE *X3*)", then

1) Case:

A) If the declared type of *X1* is CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then let *CS* be the character set of *X1*.

B) If the declared type of *X2* is CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then let *CS* be the character set of *X1*.

C) If the declared type of *X3* is CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then let *CS* be the character set of *X1*.

D) Otherwise, the character set of *CS* is undefined.

2) If *CS* is defined, then:

A) If *DP* is *X1* or *X2*, then *DT* is CHARACTER VARYING(*ML*) with character set *CS*.

B) If *DP* is *X3*, then *DT* is CHARACTER(1) with character set *CS*.

viii) If *DP* is any of *X1*, *X2*, *X3*, or X4 in a <string value function> of the form "OVERLAY <left paren> *X1* PLACING *X2* FROM *X3* FOR X4 <right paren>" or "OVERLAY <left paren> *X1* PLACING *X2* FROM *X3* <right paren>", then

Case:

1) If *DP* is *X1* (*X2*), then

Case:

A) If the declared type of *X2* (*X1*) is CHARACTER or CHARACTER VARYING with character set *CS*, *DT* is CHARACTER VARYING (*ML*) with character set *CS*.

B) Otherwise, *DT* is the declared type of *X2* (*X1*).

2) Otherwise, *DT* is NUMERIC (*MP*, 0).

ix) If *DP* is either *X1* or *X2* in a <value expression> of the form "*X1* <concatenation operator> *X2*" and *DP* is *X1* (*X2*), then

Case:

1) If the declared type of *X2* (*X1*) is CHARACTER or CHARACTER VARYING with character set *CS*, then *DT* is CHARACTER VARYING (*ML*) with character set *CS*.

2) Otherwise, *DT* is the declared type of *X2* (*X1*).

x) If *DP* is either *X1* or *X2* in a <value expression> of the form "*X1* <asterisk> *X2*" or "*X1* <solidus> *X2*" and *DP* is *X1* (*X2*), then

Case:

1) If *DP* is *X1*, then *DT* is the declared type of *X2*.

2) Otherwise,

Case:

A) If the declared type of *X1* is an interval type, then *DT* is NUMERIC (*MP*, 0).

B) Otherwise, *DT* is the declared type of *X2* (*X1*).

xi) If *DP* is either *X1* or *X2* in a <value expression> of the form "*X1* <plus sign> *X2*" or "*X1* <minus sign> *X2*", then

Case:

1) If *DP* is *X1* in an expression of the form "*X1* <minus sign> *X2*", then *DT* is the declared type of *X2*.

2) Otherwise, if *DP* is *X1* (*X2*), then

Case:

A) If the declared type of *X2* (*X1*) is date, then *DT* is INTERVAL YEAR (*PR*) TO MONTH, where *PR* is the implementation-defined maximum <interval leading field precision>.

B) If the declared type of *X2* (*X1*) is time or timestamp, then *DT* is INTERVAL DAY (*PR*) TO SECOND(*FR*), where *PR* and *FR* are the implementation-defined maximum <interval leading field precision> and maximum <interval fractional seconds precision>, respectively.

C) Otherwise, *DT* is the declared type of *X2* (*X1*).

xii) If *DP* is the <value expression primary> simply contained in a <boolean primary>, then *DT* is BOOLEAN.

xiii) If *DP* is an <array element> simply contained in an <array element list> *AEL* or *DP* represents the value of a subfield *SF* of the declared type of an <array element> simply contained in an <array element list> *AEL*, then let *ET* be the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <array element>s simply contained in *AEL*.

Case:

1) If *DP* is an <array element> of *AEL*, then *DT* is *ET*.

2) Otherwise, *DT* is the declared type of the subfield of *ET* that corresponds to *SF*.

xiv) If *DP* is a <multiset element> simply contained in a <multiset element list> *MEL* or *DP* represents the value of a subfield *SF* of the declared type of a <multiset element> simply contained in a <multiset element list> *MEL*, then let *ET* be the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <multiset element>s simply contained in *MEL*.

Case:

1) If *DP* is a <multiset element> of *MEL*, then *DT* is *ET*.

2) Otherwise, *DT* is the declared type of the subfield of *ET* that corresponds to *SF*.

xv) If *DP* is the <cast operand> simply contained in a <cast specification> *CS* or *DP* represents the value of a subfield *SF* of the declared type of the <cast operand> simply contained in a <cast specification> *CS*, then let *CT* be the simply contained <cast target> of *CS*.

Case:

1) Let *RT* be a data type determined as follows:

Case:

A) If *CT* immediately contains ARRAY or MULTISET, then *RT* is undefined.

B) If *CT* immediately contains <data type>, then *RT* is that data type.

C) If *CT* simply contains <domain name> *D*, then *RT* is the declared type of the domain identified by *D*.

2) Case:

A) If *DP* is the <cast operand> of *CS*, *DT* is *RT*.

B) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xvi) If *DP* is a <value expression> simply contained in a <case abbreviation> *CA* or *DP* represents the value of a subfield *SF* of the declared type of such a <value expression>, then let *RT* be the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <value expression>s simply contained in *CA*.

Case:

1) If *DP* is a <value expression> simply contained in *CA*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xvii) If *DP* is a <result expression> simply contained in a <case specification> *CE* or *DP* represents the value of a subfield *SF* of the declared type of such a <result expression>, then let *RT* be the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <result expression>s simply contained in *CE*.

Case:

1) If *DP* is a <result expression> simply contained in *CE*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xviii) If *DP* is a <case operand> or <when operand> simply contained in a <simple case> *CE* or *DP* represents the value of a subfield *SF* of the declared type of such a <case operand> or <when operand>, then *RT* is the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <case operand> and <when operand>s simply contained in *CE*.

Case:

1) If *DP* is a <case operand> or <when operand> simply contained in *CE*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xix) If *DP* is a <row value expression> or <contextually typed row value expression> simply contained in a <table value constructor> or <contextually typed table value constructor> *TVC*, or if *DP* represents the value of a subfield *SF* of the declared type of such a <row value expression> or <contextually typed row value expression>, then

Case:

1) Let *RT* be a data type determined as follows.

Case:

A) If *TVC* is simply contained in a <query expression> that is simply contained in an <insert statement> *IS* or if *TVC* is immediately contained in the <insert columns and source> of an <insert statement> *IS*, then *RT* is a row type in which the declared type of the *i*-th field is the declared type of the *i*-th column in the explicit or implicit <insert column list> of *IS* and the degree of *RT* is equal to the number of columns in the explicit or implicit <insert column list> of *IS*.

B) Otherwise, *RT* is the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <row value expression>s or <contextually typed row value expression>s simply contained in *TVC*.

2) Case:

A) If *DP* is a <row value expression> or <contextually typed row value expression> simply contained in *TVC*, then *DT* is *RT*.

B) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xx) If *DP* is the <value expression> simply contained in an <merge insert value list> of an <merge insert specification> *MIS* of a <merge statement> or if *DP* represents the value of a subfield *SF* of the declared type of such a <value expression>, then let *RT* be the data type indicated in the column descriptor for the positionally corresponding column in the explicit or implicit <insert column list> contained in *MIS*.

Case:

1) If *DP* is the <value expression> simply contained in *MIS*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxi) If *DP* is a <row value predicand> simply contained in a <comparison predicate>, <distinct predicate> or <between predicate> *PR* or if *DP* represents the value of a subfield *SF* of the declared type of such a <row value predicand>, then let *RT* be the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <row value predicand>s simply contained in *PR*.

Case:

1) If *DP* is a <row value predicand> simply contained in *PR*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxii) If *DP* is a <row value predicand> simply contained in a <quantified comparison predicate> or <match predicate> *PR* or *DP* represents the value of a subfield *SF* of the declared type of such a <row value predicand>, then let *RT* be the declared type of the <table subquery> simply contained in *PR*.

Case:

1) If *DP* is a <row value predicand> simply contained in *PR*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxiii) If *DP* is a <row value predicand> simply contained in an <in predicate> *PR* or if *DP* represents the value of a subfield *SF* of the declared type of such a <row value predicand>, then let *RT* be the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the <row value predicand>s simply contained in *PR* and the declared row type of the <table subquery> (if any) simply contained in *PR*.

Case:

1) If *DP* is a <row value predicand> simply contained in *PR*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxiv) If *DP* is the first <row value constructor element> simply contained in either <row value predicand 1> *RV1* or <row value predicand 2> *RV2* in an <overlaps predicate> *PR*, then

Case:

1) If both *RV1* and *RV2* simply contain a <row value constructor predicand> whose first <row value constructor element> meets the criteria for *DPV*, then *DT* is TIMESTAMP WITH TIME ZONE.

2) Otherwise, if *DP* is simply contained in *RV1* (*RV2*), then *DT* is the declared type of the first field of *RV2* (*RV1*).

xxv) If *DP* is simply contained in a <character like predicate>, <octet like predicate>, or <similar predicate> *PR*, then let *X1* represent the <row value predicand> immediately contained in *PR*, let *X2* represent the <character pattern>, the <octet pattern> or the <similar pattern>, and let *X3* represent the <escape character> or the <escape octet>.

Case:

1) If all *X1*, *X2* and *X3* meet the criteria for *DPV*, then *DT* is CHARACTER VARYING (*ML*) with an implementation-defined character set.

2) Otherwise, let *RT* be the result of applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of *X1*, *X2* and *X3*.

Case:

A) If *RT* is CHARACTER or CHARACTER VARYING with character set *CS*, then *DT* is CHARACTER VARYING(*ML*) with character set *CS*.

B) Otherwise, *DT* is *RT*.

xxvi) If *DP* is the <value expression> simply contained in an <update source> of a <set clause> *SC* or if *DP* represents the value of a subfield *SF* of the declared type of such a <value expression>, then let *RT* be the declared type of the <update target> or <mutated set clause> specified in *SC*.

Case:

1) If *DP* is the <value expression> simply contained in *SC*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxvii) If *DP* is a <contextually typed row value expression> simply contained in a <multiple column assignment> *MCA* of a <set clause> *SC* or if *DP* represents the value of a subfield *SF* of the declared type of such a <contextually typed row value expression>, then let *RT* be a row type in which the declared type of the *i*-th field is the declared type of the <update target> or <mutated set clause> immediately contained in the *i*-th <set target> contained in the <set target list> of *MCA*.

Case:

1) If *DP* is a <contextually typed row value expression> simply contained in *MCA*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxviii) If *DP* is the <value specification> immediately contained in a <catalog name characteristic>, <schema name characteristic>, <character set name characteristic>, <SQL-path characteristic>, <transform group characteristic>, <role specification> or <set session user identifier statement>, then *DT* is CHARACTER VARYING (*ML*) with an implementation-defined character set.

xxix) If *DP* is the <interval value expression> immediately contained in a <set local time zone statement>, then *DT* is INTERVAL HOUR TO MINUTE.

xxx) If *DP* is an <SQL argument> of a <routine invocation> *RI* or if *DP* is the value of a subfield *SF* of the declared type of a <value expression> immediately contained in such an <SQL argument>, and if *DP* is the *i*-th <SQL argument> of *RI* or is contained in the *i*-th <SQL argument> of *RI*, then let *RT* denote the declared type of the *i*-th SQL parameter of the subject routine of *RI* determined by applying the Syntax Rules of Subclause 10.4, "<routine invocation>", to *RI*.

Case:

1) If *DP* is the *i*-th <SQL argument> of *RI*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxxi) If *DP* is contained in a <window frame preceding> or a <window frame following> contained in a <window specification> *WS*, then

Case:

1) If *WS* specifies ROWS, then *DT* is NUMERIC (*MP*, 0).

2) Otherwise, let *SDT* be the data type of the single <sort key> contained in *WS*.

Case:

A) If *SDT* is a numeric type, then *DT* is *SDT*.

B) If *SDT* is DATE, then *DT* is INTERVAL DAY.

C) If *SDT* is TIME(*P*) WITHOUT TIME ZONE or TIME(*P*) WITH TIME ZONE, then *DT* is INTERVAL HOUR TO SECOND(*P*).

D) If *SDT* is TIMESTAMP(*P*) WITHOUT TIME ZONE or TIMESTAMP(*P*) WITH TIME ZONE, then *DT* is INTERVAL DAY TO SECOND(*P*).

E) If *SDT* is an interval type, then *DT* is *SDT*.

xxxii) If *DP* is a <locator reference> simply contained in a <hold locator statement> or a <free locator statement>, then *DT* is INTEGER.

b) If *DT* is undefined, then an exception condition is raised: *syntax error or access rule violation*.

7) Whether a <dynamic parameter specification> is an input argument, an output argument, or both an input and an output argument is determined as follows.

Case:

a) If *P* is a <call statement>, then:

i) Let *SR* be the subject routine of the <routine invocation> *RI* immediately contained in *P*. Let *n* be the number of <SQL argument>s in the <SQL argument list> immediately contained in *RI*.

ii) Let A_y, 1 (one) $\le y \le n$, be the *y*-th <SQL argument> of the <SQL argument list> immediately contained in *RI*.

iii) For each <dynamic parameter specification> *D* contained in some <SQL argument> A_k, 1 (one) $\le k \le n$:

1) *D* is an input <dynamic parameter specification> if the <parameter mode> of the *k*-th SQL parameter of *SR* of *SR* is IN or INOUT.

2) *D* is an output <dynamic parameter specification> if the <parameter mode> of the *k*-th SQL parameter of *SR* is OUT or INOUT.

b) Otherwise:

i) If a <dynamic parameter specification> is contained in a <target specification>, then it is an output <dynamic parameter specification>.

ii) If a <dynamic parameter specification> is contained in a <value specification>, then it is an input <dynamic parameter specification>.

8) If *P* or *PS* is a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>, then let *CN* be the <cursor name> contained in *P* or *PS*, respectively.

Case:

a) If *P* or *PS* contains a <scope option> that specifies GLOBAL, then

Case:

i) If there exists an extended dynamic cursor *EDC* with an <extended cursor name> having a global scope and a <cursor name> that is equivalent to *CN*, then *EDC* is the cursor referenced by *P* or *PS*.

ii) Otherwise, an exception condition is raised: *invalid cursor name*.

b) If *P* or *PS* contains a <scope option> that specifies LOCAL, or if no <scope option> is specified, then the *potentially referenced cursors* of *P* or *PS* include every declared dynamic cursor whose <cursor name> is equivalent to *CN* and whose scope is the containing module and every extended dynamic cursor having an <extended cursor name> that has a scope of the containing module and whose <cursor name> is equivalent to *CN*.

Case:

i) If the number of potentially referenced cursors is greater than 1 (one), then an exception condition is raised: *ambiguous cursor name*.

ii) If the number of potentially referenced cursors is less than 1 (one), then an exception condition is raised: *invalid cursor name*.

iii) Otherwise, *CN* refers to the single potentially referenced cursor of *P*.

9) If <extended statement name> is specified for the <SQL statement name>, then let *S* be <simple value specification> and let *V* be the character string that is the result of

```
TRIM ( BOTH ' ' FROM S )
```

If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid SQL statement identifier*.

10) If <statement name> is specified for the <SQL statement name>, *P* is not a <cursor specification>, and <statement name> is associated with a cursor *C* through a <dynamic declare cursor>, then an exception condition is raised: *dynamic SQL error — prepared statement not a cursor specification*.

11) If the value of the <SQL statement name> identifies an existing prepared statement, then an implicit

```
DEALLOCATE PREPARE SSN
```

is executed, where *SSN* is the value of the <SQL statement name>.

12) *P* is prepared for execution, resulting in a prepared statement *PRP*.

Case:

a) If the <prepare statement> is contained in an <SQL routine> *R*, then

Case:

i) If the security characteristic of *R* is DEFINER, then the owner of *PRP* is set to the owner of *R*.

ii) Otherwise, *PRP* has no owner.

b) If the <prepare statement> is contained in a triggered action, then the owner of *PRP* is set to the owner of the trigger.

c) Otherwise,

NOTE 429 — If the <prepare statement> is in neither of the above, then it must necessarily be immediately contained in an externally-invoked procedure.

Case:

i) If the SQL-client module that includes the <prepare statement> has a <module authorization identifier> *MAI* and FOR STATIC ONLY was not specified in the <SQL-client module definition>, then the owner of *PRP* is *MAI*.

ii) Otherwise, *PRP* has no owner.

13) Case:

a) If <extended statement name> is specified for the <SQL statement name>, then the value of the <extended statement name> is associated with the prepared statement. This value and explicit or implied <scope option> shall be specified for each <execute statement> or <allocate cursor statement> that is to be associated with this prepared statement.

b) If <statement name> is specified for the <SQL statement name>, then:

i) If <statement name> is not associated with a cursor and either *P* is not a <cursor specification> or *P* is a <cursor specification> that conforms to the Format and Syntax Rules of a <dynamic single row select statement>, then an equivalent <statement name> shall be specified for each <execute statement> that is to be associated with this prepared statement.

ii) If *P* is a <cursor specification> and <statement name> is associated with a cursor *C* through a <dynamic declare cursor>, then an association is made between *C* and *P*. The association is preserved until the prepared statement is destroyed.

14) The validity of an <extended statement name> value or a <statement name> that does not identify a held cursor in an SQL-transaction different from the one in which the statement was prepared is implementation-dependent.

15) If <attributes specification> is specified, then let *ATV* be the contents of the <attributes variable>. If *ATV* is not a zero-length character string, then

a) If *ATV* does not conform to the Format and Syntax Rules of Subclause 19.7, "<cursor attributes>", then an exception condition is raised: *syntax error or access rule violation*.

b) Let *N* be the number of <dynamic declare cursor>s in the containing <SQL-client module definition> whose <statement name> is equivalent to the <statement name> of the <prepare statement>.

c) If $N > 0$ (zero), then let CR_i, 1 (one) $\le i \le N$, be the cursor specified by the *i*-th <dynamic declare cursor> in the containing <SQL-client module definition>. For 1 (one) $\le i \le N$:

i) If *ATV* includes <cursor sensitivity> *CS*, then the sensitivity of CR_i is set to *CS*.

ii) If *ATV* includes <cursor scrollability> *CL*, then the scrollability of CR_i is set to *CL*.

iii) If *ATV* includes <cursor holdability> *CH*, then the holdability of CR_i is set to *CH*.

iv) If *ATV* includes <cursor returnability> *CR*, then the returnability of CR_i is set to *CR*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <prepare statement>.

2) Without Feature B034, "Dynamic specification of cursor attributes", conforming SQL language shall not contain an <attributes specification>.

19.7 <cursor attributes>

Function

Specify a list of cursor attributes.

Format

```
<cursor attributes> ::= <cursor attribute>...

<cursor attribute> ::=
    <cursor sensitivity>
  | <cursor scrollability>
  | <cursor holdability>
  | <cursor returnability>
```

Syntax Rules

1) Each of <cursor sensitivity>, <cursor scrollability>, <cursor holdability> and <cursor returnability> shall be specified at most once.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

19.8 <deallocate prepared statement>

Function

Deallocate SQL-statements that have been prepared with a <prepare statement>.

Format

```
<deallocate prepared statement> ::= DEALLOCATE PREPARE <SQL statement name>
```

Syntax Rules

1) If <SQL statement name> is a <statement name>, then the containing <SQL-client module definition> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <deallocate prepared statement>.

Access Rules

None.

General Rules

1) If the <SQL statement name> does not identify a statement prepared in the scope of the <SQL statement name>, then an exception condition is raised: *invalid SQL statement name*.

2) If the value of <SQL statement name> identifies an existing prepared statement that is the <cursor specification> of an open cursor, then an exception condition is raised: *invalid cursor state*.

3) The prepared statement identified by the <SQL statement name> is destroyed. Any cursor that was allocated with an <allocate cursor statement> that is associated with the prepared statement identified by the <SQL statement name> is destroyed. If the value of the <SQL statement name> identifies an existing prepared statement that is a <cursor specification>, then any prepared statements that reference that cursor are destroyed.

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <deallocate prepared statement>.

19.9 <describe statement>

Function

Obtain information about the <select list> columns or <dynamic parameter specification>s contained in a prepared statement or about the columns of the result set associated with a cursor.

Format

```
<describe statement> ::=
    <describe input statement>
  | <describe output statement>

<describe input statement> ::=
    DESCRIBE INPUT <SQL statement name> <using descriptor> [ <nesting option> ]

<describe output statement> ::=
    DESCRIBE [ OUTPUT ] <described object> <using descriptor> [ <nesting option> ]

<nesting option> ::=
    WITH NESTING
  | WITHOUT NESTING

<using descriptor> ::= USING [ SQL ] DESCRIPTOR <descriptor name>

<described object> ::=
    <SQL statement name>
  | CURSOR <extended cursor name> STRUCTURE
```

Syntax Rules

1) If <SQL statement name> is a <statement name>, then the containing <SQL-client module definition> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <describe statement>.

2) If <nesting option> is not specified, then WITHOUT NESTING is implicit.

Access Rules

None.

General Rules

1) If <describe input statement> is executed and the value of the <SQL statement name> does not identify a statement prepared in the scope of the <SQL statement name>, then an exception condition is *invalid SQL statement name*.

2) If <describe output statement> is executed, <SQL statement name> is specified, and the value of the <SQL statement name> does not identify a statement prepared in the scope of the <SQL statement name>, then an exception condition is *invalid SQL statement name*.

3) If <describe output statement> is executed, <extended cursor name> is specified, and the value of the <extended cursor name> does not identify a known allocated cursor, then an exception condition is *invalid cursor name*.

4) If an SQL system descriptor area is not currently allocated whose name is the value of the <simple value specification> immediately contained in <descriptor name> and whose scope is specified by the <scope option> immediately contained in <descriptor name>, then an exception condition is raised: *invalid SQL descriptor name*.

5) Let *DA* be the descriptor area identified by <descriptor name>. Let *N* be the <occurrences> specified when *DA* was allocated.

6) Case:

   a) If the statement being executed is a <describe input statement>, then a descriptor for the input <dynamic parameter specification>s for the prepared statement is stored in *DA*. Let *D* be the number of input <dynamic parameter specification>s in the prepared statement. If WITH NESTING is specified, then let NS_i, 1 (one) $\le i \le D$, be the number of subordinate descriptors of the descriptor for the i-th input dynamic parameter; otherwise, let NS_i be 0 (zero).

   b) If the statement being executed is a <describe output statement> and the prepared statement that is being described is a <dynamic select statement> or a <dynamic single row select statement>, then a descriptor for the <select list> columns for the prepared statement is stored in *DA*. Let *T* be the table defined by the prepared statement and let *D* be the degree of *T*. If WITH NESTING is specified, then let NS_i, 1 (one) $\le i \le D$, be the number of subordinate descriptors of the descriptor for the i-th column of *T*; otherwise, let NS_i be 0 (zero).

   c) Otherwise, a descriptor for the output <dynamic parameter specification>s for the prepared statement is stored in *DA*. Let *D* be the number of output <dynamic parameter specification>s in the prepared statement. If WITH NESTING is specified, then let NS_i, 1 (one) $\le i \le D$, be the number of subordinate descriptors of the descriptor for the i-th output dynamic parameter; otherwise, let NS_i be 0 (zero).

7) *DA* is set as follows:

   a) Let *TD* be the value of $D+NS_1+NS_2+...+NS_D$. COUNT is set to *TD*.

   b) TOP_LEVEL_COUNT is set to *D*.

   c) DYNAMIC_FUNCTION and DYNAMIC_FUNCTION_CODE are set to the identifier and code, respectively, for the prepared statement as shown in Table 31, "SQL-statement codes".

   d) If the statement being executed is a <describe output statement> and the prepared statement that is being described is a <dynamic select statement> or a <dynamic single row select statement>:

      Case:

      i) If some subset of the columns of *T* is the primary key of *T*, then KEY_TYPE is set to 1 (one).

ii) If some subset of the columns of *T* is the preferred candidate key of *T*, then KEY_TYPE is set to 2.

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

NOTE 430 — Primary keys and preferred candidate keys are defined in Subclause 4.18, "Functional dependencies".

e) If *TD* is greater than *N*, then a completion condition is raised: *warning — insufficient item descriptor areas*.

f) If *TD* is 0 (zero) or *TD* is greater than *N*, then no item descriptor areas are set. Otherwise:

i) The first *TD* item descriptor areas are set with values from the descriptors and, optionally, subordinate descriptors for

Case:

1) If the statement being executed is a <describe input statement>, then the input <dynamic parameter specification>s.

2) If the statement being executed is a <describe output statement> and the statement being described is a <dynamic select statement> or a <dynamic single row select statement>, then the columns of *T*.

3) Otherwise, the output <dynamic parameter specification>s.

ii) The descriptor for the first such column or <dynamic parameter specification> is assigned to the first item descriptor area.

iii) If the descriptor for the *j*-th column or <dynamic parameter specification> is assigned to the *k*-th item descriptor area, then:

1) The descriptor for the (*j*+1)-th column or <dynamic parameter specification> is assigned to the ($k+NS_{j+1}$)-th item descriptor area.

2) If WITH NESTING is specified, then the implicitly ordered subordinate descriptors for the *j*-th column or <dynamic parameter specification> are assigned to contiguous item descriptor areas starting at the (*k*+1)-th item descriptor area.

8) An SQL item descriptor area, if set, consists of values for LEVEL, TYPE, NULLABLE, NAME, UNNAMED, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, PARAMETER_SPECIFIC_NAME, and other fields depending on the value of TYPE as described below. The DATA and INDICATOR fields are not relevant. Those fields and fields that are not applicable for a particular value of TYPE are set to implementation-dependent values.

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

b) TYPE is set to a code, as shown in Table 25, "Codes used for SQL data types in Dynamic SQL", indicating the declared type of the column, <dynamic parameter specification>, or subordinate descriptor.

c) Case:

i) If the value of LEVEL is 0 (zero) and the item descriptor area describes a column, then:

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

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

3) If the column is a member of the primary key of *T* and KEY_TYPE was set to 1 (one) or if the column is a member of the preferred candidate key of *T* and KEY_TYPE was set to 2, then KEY_MEMBER is set to 1 (one); otherwise, KEY_MEMBER is set to 0 (zero).

ii) If the value of LEVEL is 0 (zero) and the item descriptor area describes a <dynamic parameter specification>, then:

1) NULLABLE is set to 1 (one).

NOTE 431 — This indicates that the <dynamic parameter specification> can have the null value.

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

3) KEY_MEMBER is set to 0 (zero).

iii) Otherwise:

1) NULLABLE is set to 1 (one).

2) Case:

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

Case:

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

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

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

3) KEY_MEMBER is set to 0 (zero).

d) Case:

i) If TYPE indicates a <character string type>, then:

1) LENGTH is set to the length or maximum length in characters of the character string type.

2) OCTET_LENGTH is set to the maximum possible length in octets of the character string type.

3) CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are set to the the fully qualified name of the character string type's character set.

4) COLLATION_CATALOG, COLLATION_SCHEMA and COLLATION_NAME are set to the fully qualified name of the character string type's declared type collation, if any, and otherwise to the empty string.

If the subject <language clause> specifies C, then the lengths specified in LENGTH and OCTET_LENGTH do not include the implementation-defined null character that terminates a C character string.

ii) If TYPE indicates a <binary large object string type>, then LENGTH is set to the length or maximum length in octets of the binary string and OCTET_LENGTH is set to the maximum possible length in octets of the binary string.

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

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

v) If TYPE indicates a <datetime type>, then LENGTH is set to the length in positions of the datetime type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 26, "Codes associated with datetime data types in Dynamic SQL", to indicate the specific datetime data type and PRECISION is set to the <time precision> or <timestamp precision>, if either is applicable.

vi) If TYPE indicates an <interval type>, then LENGTH is set to the length in positions of the interval type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 27, "Codes used for <interval qualifier>s in Dynamic SQL", to indicate the <interval qualifier> of the interval data type, DATETIME_INTERVAL_PRECISION is set to the <interval leading field precision> and PRECISION is set to the <interval fractional seconds precision>, if applicable.

vii) If TYPE indicates a user-defined type, then USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the fully qualified name of the user-defined type, and USER_DEFINED_TYPE_CODE is set to a code as specified in Table 29, "Codes associated with user-defined types in Dynamic SQL", to indicate the category of the user-defined type.

viii) If TYPE indicates a <reference type>, then:

1) USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the fully qualified name of the referenced type.

2) SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME are set to the fully qualified name of the referenceable base table.

3) LENGTH and OCTET_LENGTH are set to the length in octets of the <reference type>.

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

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

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

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

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

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

NOTE 432 — A P whose <parameter mode> is IN can be a <value expression> that contains zero, one, or more <dynamic parameter specification>s. Thus:

— Every D_x maps to one and only one P_y.

— Several D_x instances can map to the same P_y.

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

iv) The PARAMETER_MODE value in the descriptor for each D_x is set to the value from Table 28, "Codes used for input/output SQL parameter modes in Dynamic SQL", that indicates the <parameter mode> of P_y.

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

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

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <describe input statement>.

2) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <describe output statement>.

19.10 <input using clause>

Function

Supply input values for an <SQL dynamic statement>.

Format

```
<input using clause> ::=
    <using arguments>
  | <using input descriptor>

<using arguments> ::= USING <using argument> [ { <comma> <using argument> }... ]

<using argument> ::= <general value specification>

<using input descriptor> ::= <using descriptor>
```

Syntax Rules

1) The <general value specification> immediately contained in <using argument> shall be either a <host parameter specification> or an <embedded variable specification>.

Access Rules

None.

General Rules

1) If <using input descriptor> is specified and an SQL descriptor area is not currently allocated whose name is the value of the <simple value specification> immediately contained in <descriptor name> and whose scope is specified by the <scope option> immediately contained in <descriptor name>, then an exception condition is raised: *invalid SQL descriptor name*.

2) When an <input using clause> is used in a <dynamic open statement> or as the <parameter using clause> in an <execute statement>, the <input using clause> describes the input <dynamic parameter specification> values for the <dynamic open statement> or the <execute statement>, respectively. Let *PS* be the prepared <dynamic select statement> referenced by the <dynamic open statement> or the prepared statement referenced by the <execute statement>, respectively.

3) Let *D* be the number of input <dynamic parameter specification>s in *PS*.

4) If <using arguments> is specified and the number of <using argument>s is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

5) If <using input descriptor> is specified, then:

   a) Let *N* be the value of COUNT.

b) If *N* is greater than the value of <occurrences> specified when the SQL descriptor area identified by <descriptor name> was allocated or is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count.*

c) If the first *N* item descriptor areas are not valid as specified in Subclause 19.1, "Description of SQL descriptor areas", then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications.*

d) In the first *N* item descriptor areas:

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

ii) If the value of INDICATOR is not negative, TYPE does not indicate ROW, and the item descriptor area is not subordinate to an item descriptor area whose INDICATOR value is negative or whose TYPE field indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, and if the value of DATA is not a valid value of the data type represented by the item descriptor area, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications.*

6) For 1 (one) ≤ *i* ≤ *D*:

a) Let *TDT* be the effective declared type of the *i*-th input <dynamic parameter specification>, defined to be the type represented by the item descriptor area and its subordinate descriptor areas that would be set by a <describe input statement> to reflect the description of the *i*-th input <dynamic parameter specification> of *PS*.

NOTE 433 — See the General Rules of Subclause 19.9, "<describe statement>".

NOTE 434 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 19.1, "Description of SQL descriptor areas".

b) Case:

i) If <using input descriptor> is specified, then:

1) Let *IDA* be the *i*-th item descriptor area whose LEVEL value is 0 (zero).

2) Let *SDT* be the effective declared type represented by *IDA*.

NOTE 435 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 19.1, "Description of SQL descriptor areas".

3) Let *SV* be the *associated value* of *IDA*.

Case:

A) If the value of INDICATOR is negative, then *SV* is the null value.

B) Otherwise,

Case:

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

II) Otherwise, *SV* is the value of DATA with data type *SDT*.

ii) If <using arguments> is specified, then let *SDT* and *SV* be the declared type and value, respectively, of the *i*-th <using argument>.

c) Case:

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

Case:

1) If *SV* is not the null value, then let the value of the *i*-th dynamic parameter be the value of *SV*.

2) Otherwise, let the value of the *i*-th dynamic parameter be the null value.

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

Case:

1) If the <cast specification>

```
CAST ( IV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", and there is an implementation-defined conversion from type *STD* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *IV* to type *TDT*, and the result is the value *TV* of the *i*-th input dynamic parameter.

2) Otherwise:

A) If the <cast specification>

```
CAST ( IV AS TDT )
```

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

B) If the <cast specification>

```
CAST ( IV AS TDT )
```

does not conform to the General Rules of Subclause 6.12, "<cast specification>", then an exception condition is raised in accordance with the General Rules of Subclause 6.12, "<cast specification>".

C) The <cast specification>

```
CAST ( IV AS TDT )
```

is effectively performed and is the value of the *i*-th input dynamic parameter.

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

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

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

3) The Syntax Rules of Subclause 9.19, "Determination of a to-sql function", are applied with *DT* and *GN* as *TYPE* and *GROUP*, respectively.

   Case:

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

      Case:

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

         Case:

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

            ```
            DT()
            ```

            as the first parameter and *SV* as the second parameter. The <return value> is the value of the *i*-th input dynamic parameter.

         2) Otherwise, *TSF* is effectively invoked with *SV* as the first parameter. The <return value> is the value of the *i*-th input dynamic parameter.

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

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

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <input using clause>.

19.11 <output using clause>

Function

Supply output variables for an <SQL dynamic statement>.

Format

```
<output using clause> ::=
    <into arguments>
  | <into descriptor>

<into arguments> ::= INTO <into argument> [ { <comma> <into argument> }... ]

<into argument> ::= <target specification>

<into descriptor> ::= INTO [ SQL ] DESCRIPTOR <descriptor name>
```

Syntax Rules

1) The <target specification> immediately contained in <into argument> shall be either a <host parameter specification> or an <embedded variable specification>.

Access Rules

None.

General Rules

1) If <into descriptor> is specified and an SQL descriptor area is not currently allocated whose name is the value of the <simple value specification> immediately contained in <descriptor name> and whose scope is specified by the <scope option> immediately contained in <descriptor name>, then an exception condition is raised: *invalid SQL descriptor name*.

2) When an <output using clause> is used in a <dynamic fetch statement> or as the <result using clause> of an <execute statement>, let *PS* be the prepared <dynamic select statement> referenced by the <dynamic fetch statement> or the prepared <dynamic single row select statement> referenced by the <execute statement>, respectively.

3) Case:

   a) If *PS* is a <dynamic select statement> or a <dynamic single row select statement>, then the <output using clause> describes the <target specification>s for the <dynamic fetch statement> or the <execute statement>. Let *D* be the degree of the table specified by *PS*.

   b) Otherwise, the <output using clause> describes the <target specification>s for the output <dynamic parameter specification>s contained in *PS*. Let *D* be the number of such output <dynamic parameter specification>s.

4) If <into arguments> is specified and the number of <into argument>s is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications*.

5) If <into descriptor> is specified, then:

   a) Let *N* be the value of COUNT.

   b) If *N* is greater than the value of <occurrences> specified when the SQL descriptor area identified by <descriptor name> was allocated or less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count*.

   c) If the first *N* item descriptor areas are not valid as specified in Subclause 19.1, "Description of SQL descriptor areas", then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications*.

   d) In the first *N* item descriptor areas, if the number of item descriptor areas in which the value of LEVEL is 0 (zero) is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications*.

6) For 1 (one) ≤ *i* ≤ *D*:

   a) Let *SDT* be the effective declared type of the *i*-th <select list> column or output <dynamic parameter specification>, defined to be the type represented by the item descriptor area and its subordinate descriptor areas that would be set by

      Case:

      i) If *PS* is a <dynamic select statement> or a <dynamic single row select statement>, then a <describe output statement> to reflect the description of the *i*-th <select list> column; let *SV* be the value of that <select list> column, with data type *SDT*.

      ii) Otherwise, a <describe output statement> to reflect the description of the *i*-th output <dynamic parameter specification>; let *SV* be the value of that <dynamic parameter specification>, with data type *SDT*.

      NOTE 436 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 19.1, "Description of SQL descriptor areas".

   b) Case:

      i) If <into descriptor> is specified, then let *TDT* be the declared type of the *i*-th <target specification> as represented by the *i*-th item descriptor area *IDA* whose LEVEL value is 0 (zero).

      NOTE 437 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 19.1, "Description of SQL descriptor areas".

      ii) If <into arguments> is specified, then let *TDT* be the data type of the *i*-th <into argument>.

   c) If the <output using clause> is used in a <dynamic fetch statement>, then let *LTDT* be the data type on the most recently executed <dynamic fetch statement>, if any, for the cursor *CR*. It is implementation-defined whether or not an exception condition is raised: *dynamic SQL error — restricted data type attribute violation* if any of the following are true:

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

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

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

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

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

d) Case:

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

Case:

1) If *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and is the value *TV* of the *i*-th <target specification>.

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

ii) If *STD* and *TDT* are predefined data types, then

Case:

1) If the <cast specification>

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

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

2) Otherwise:

A) If the <cast specification>

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

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

B) If the <cast specification>

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

does not conform to the General Rules of Subclause 6.12, "<cast specification>", then an exception condition is raised in accordance with the General Rules of Subclause 6.12, "<cast specification>".

C) The <cast specification>

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

is effectively performed, and is the value *TV* of the *i*-th <target specification>.

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

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

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

3) Apply the Syntax Rules of Subclause 9.17, "Determination of a from-sql function", with *DT* and *GN* as *TYPE* and *GROUP*, respectively.

Case:

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

Case:

I) If *FSFRT* is compatible with *TDT*, then the from-sql function *FSF* is effectively invoked with *SV* as its input SQL parameter and the <return value> is the value *TV* of the *i*-th <target specification>.

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

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

e) Case:

i) If <into descriptor> is specified, then *IDA* is set to reflect the value of *TV* as follows:

Case:

1) If TYPE indicates ROW, then

Case:

A) If *TV* is the null value, then the value of INDICATOR in *IDA* and in all subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR is set to –1.

B) Otherwise, the *i*-th subordinate descriptor area of *IDA* is set to reflect the value of the *i*-th field of *TV* by applying this subrule (beginning with the outermost 'Case') to the *i*-th subordinate descriptor area of *IDA* as *IDA*, the value of the *i*-th field of *TV* as *TV*, the value of the *i*-th field of *SV* as *SV*, and the data type of the *i*-th field of *SV* as *SDT*.

2) Otherwise,

Case:

A) If *TV* is the null value, then the value of INDICATOR is set to –1.

B) If *TV* is not the null value, then:

I) The value of INDICATOR is set to 0 (zero).

II) Case:

1) If TYPE indicates a locator type, then a locator *L* that uniquely identifies *TV* is generated and the value of DATA is set to an implementation-dependent four-octet value that represents *L*.

2) Otherwise, the value of DATA is set to *TV*.

III) Case:

1) If TYPE indicates CHARACTER VARYING or BINARY LARGE OBJECT, then RETURNED_LENGTH is set to the length in characters or octets, respectively, of *TV*, and RETURNED_OCTET_LENGTH is set to the length in octets of *TV*.

2) If *SDT* is CHARACTER VARYING or BINARY LARGE OBJECT, then RETURNED_LENGTH is set to the length in characters or octets, respectively, of *SV*, and RETURNED_OCTET_LENGTH is set to the length in octets of *SV*.

3) If TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then RETURNED_CARDINALITY is set to the cardinality of *TV*.

ii) If <into arguments> is specified, then the Rules in Subclause 9.1, "Retrieval assignment", are applied to *TV* and the *i*-th <into argument> as *VALUE* and *TARGET*, respectively.

NOTE 438 — All other values of the SQL descriptor area are unchanged.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <output using clause>.

19.12 <execute statement>

Function

Associate input SQL parameters and output targets with a prepared statement and execute the statement.

Format

```
<execute statement> ::=
    EXECUTE <SQL statement name> [ <result using clause> ] [ <parameter using clause> ]

<result using clause> ::= <output using clause>

<parameter using clause> ::= <input using clause>
```

Syntax Rules

1) If <SQL statement name> is a <statement name>, then the containing <SQL-client module definition> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <execute statement>.

Access Rules

None.

General Rules

1) When the <execute statement> is executed, if the <SQL statement name> does not identify a prepared statement *P*, then an exception condition is raised: *invalid SQL statement name.*

2) Let *PS* be the statement previously prepared using <SQL statement name>.

3) If *PS* is a <dynamic select statement> that does not conform to the Format and Syntax Rules of a <dynamic single row select statement>, then an exception condition is raised: *dynamic SQL error — cursor specification cannot be executed.*

4) If *PS* contains the <table name> of a created or declared local temporary table and if the <execute statement> is not in the same <SQL-client module definition> as the <prepare statement> that prepared the prepared statement, then an exception condition is raised: *syntax error or access rule violation.*

5) If *PS* contains input <dynamic parameter specification>s and a <parameter using clause> is not specified, then an exception condition is raised: *dynamic SQL error — using clause required for dynamic parameters.*

6) If *PS* is a <dynamic single row select statement> or it contains output <dynamic parameter specification>s and a <result using clause> is not specified, then an exception condition is raised: *dynamic SQL error — using clause required for result fields.*

7) If a <parameter using clause> is specified, then the General Rules specified in Subclause 19.10, "<input using clause>", for a <parameter using clause> in an <execute statement> are applied.

8) A copy of the top cell is pushed onto the authorization stack. If *PS* has an owner, then the top cell of the authorization stack is set to contain only the authorization identifier of the owner of *PS*.

9) The General Rules of Subclause 13.5, "<SQL procedure statement>", are evaluated with *PS* as the executing statement.

10) If *PS* is a <preparable dynamic delete statement: positioned>, then when it is executed all General Rules in Subclause 19.22, "<preparable dynamic delete statement: positioned>", apply to the <preparable statement>.

11) If *PS* is a <preparable dynamic update statement: positioned>, then when it is executed, all General Rules in Subclause 19.23, "<preparable dynamic update statement: positioned>", apply to the <preparable statement>.

12) If a <result using clause> is specified, then the General Rules specified in Subclause 19.11, "<output using clause>", for a <result using clause> in an <execute statement> are applied.

13) Upon completion of execution, the top cell in the authorization stack is removed.

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <result using clause>.

2) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <execute statement>.

19.13 <execute immediate statement>

Function

Dynamically prepare and execute a preparable statement.

Format

```
<execute immediate statement> ::=
    EXECUTE IMMEDIATE <SQL statement variable>
```

Syntax Rules

1) The declared type of <SQL statement variable> shall be character string.

Access Rules

None.

General Rules

1) Let *P* be the contents of the <SQL statement variable>.

2) If one or more of the following are true, then an exception condition is raised: *syntax error or access rule violation.*

   a) *P* is a <dynamic select statement> or a <dynamic single row select statement>.

   b) *P* contains a <dynamic parameter specification>.

3) Let *SV* be <SQL statement variable>. <execute immediate statement> is equivalent to the following:

```
PREPARE IMMEDIATE_STMT FROM SV ;
EXECUTE IMMEDIATE_STMT ;
DEALLOCATE PREPARE IMMEDIATE_STMT ;
```

   where *IMMEDIATE_STMT* is an implementation-defined <statement name> that is not equivalent to any other <statement name> in the containing <SQL-client module definition>.

   NOTE 439 — Exception condition or completion condition information resulting from the PREPARE or EXECUTE is reflected in the diagnostics area.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <execute immediate statement>.

19.14 <dynamic declare cursor>

Function

Declare a cursor to be associated with a <statement name>, which may in turn be associated with a <cursor specification>.

Format

```
<dynamic declare cursor> ::=
    DECLARE <cursor name> [ <cursor sensitivity> ] [ <cursor scrollability> ] CURSOR
    [ <cursor holdability> ]
    [ <cursor returnability> ]
    FOR <statement name>
```

Syntax Rules

1) The <cursor name> shall not be identical to the <cursor name> specified in any other <declare cursor> or <dynamic declare cursor> in the same <SQL-client module definition>.

2) The containing <SQL-client module definition> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <dynamic declare cursor>.

3) If <cursor scrollability> is not specified, then NO SCROLL is implicit.

4) If <cursor holdability> is not specified, then WITHOUT HOLD is implicit.

5) If <cursor returnability> is not specified, then WITHOUT RETURN is implicit.

Access Rules

None.

General Rules

1) All General Rules of Subclause 14.1, "<declare cursor>", apply to <dynamic declare cursor>, replacing "<cursor specification>" with "prepared statement".

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic declare cursor>.

19.15 <allocate cursor statement>

Function

Define a cursor based on a prepared statement for a <cursor specification> or assign a cursor to the ordered set of result sets returned from an SQL-invoked procedure.

Format

```
<allocate cursor statement> ::=
    ALLOCATE <extended cursor name> <cursor intent>

<cursor intent> ::=
    <statement cursor>
  | <result set cursor>

<statement cursor> ::=
    [ <cursor sensitivity> ] [ <cursor scrollability> ] CURSOR
    [ <cursor holdability> ]
    [ <cursor returnability> ]
    FOR <extended statement name>

<result set cursor> ::= FOR PROCEDURE <specific routine designator>
```

Syntax Rules

1) If <result set cursor> is specified, then the SQL-invoked routine identified by <specific routine designator> shall be an SQL-invoked procedure.

Access Rules

None.

General Rules

1) Let *S* be the <simple value specification> immediately contained in <extended cursor name>. Let *V* be the character string that is the result of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

   If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid cursor name.*

2) If the value of the <extended cursor name> is identical to the value of the <extended cursor name> of any other cursor allocated in the scope of the <extended cursor name>, then an exception condition is raised: *invalid cursor name.*

3) If <statement cursor> is specified, then:

a) When the <allocate cursor statement> is executed, if the value of the <extended statement name> does not identify a statement previously prepared in the scope of the <extended statement name>, then an exception condition is raised: *invalid SQL statement name.*

b) If the prepared statement associated with the <extended statement name> is not a <cursor specification>, then an exception condition is raised: *dynamic SQL error — prepared statement not a cursor specification.*

c) All General Rules of Subclause 14.1, "<declare cursor>", apply to <allocate cursor statement>, replacing "<open statement>" with "<dynamic open statement>" and "<cursor specification>" with "prepared statement".

d) An association is made between the value of the <extended cursor name> and the prepared statement in the scope of the <extended cursor name>. The association is preserved until the prepared statement is destroyed, at which time the cursor identified by <extended cursor name> is also destroyed.

4) If <result set cursor> is specified, then:

a) When the <allocate cursor statement> is executed, if the <specific routine designator> does not identify an SQL-invoked procedure *P* that has been previously invoked during the current SQL-session, an exception condition is raised: *invalid SQL-invoked procedure reference.*

b) If *P* did not return any result sets, then an exception condition is raised: *no data — no additional dynamic result sets returned.*

c) Let *RRS* be the ordered set of result sets returned by *P*.

d) When the <allocate cursor statement> is executed, an association is made between the <extended cursor name> and the first result set *FRS* in *RRS*. The definition of *FRS* is the definition of the <cursor specification> *CS* in *P* that created *FRS*. Let *CR* be the cursor declared by the <declare cursor> that contains *CS*.

e) Let *T* be the table specified by *CS*. *T* is the first result set returned from *P*.

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

g) Cursor *CR* is placed in the open state.

Case:

i) If *CR* is scrollable, then let *CRCN* be the <cursor name> of *CR* in *P*. The position of *CR* in *T* is before the row that would be retrieved if the following SQL-statement were executed in *P*:

```
FETCH NEXT FROM CRCN_i INTO...
```

ii) Otherwise, the position of *CR* is before the first row of *T*.

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain an <allocate cursor statement>.

19.16 <dynamic open statement>

Function

Associate input dynamic parameters with a <cursor specification> and open the cursor.

Format

```
<dynamic open statement> ::= OPEN <dynamic cursor name> [ <input using clause> ]
```

Syntax Rules

1) If <dynamic cursor name> *DCN* is a <cursor name> *CN*, then the containing <SQL-client module definition> shall contain a <dynamic declare cursor> whose <cursor name> is *CN*.

Access Rules

1) The Access Rules for the <query expression> simply contained in the prepared statement associated with the <dynamic cursor name> are applied.

General Rules

1) If <dynamic cursor name> is a <cursor name> and the <statement name> of the associated <dynamic declare cursor> is not associated with a prepared statement, then an exception condition is raised: *invalid SQL statement name*.

2) If <dynamic cursor name> is an <extended cursor name> whose value does not identify a cursor allocated in the scope of the <extended cursor name>, then an exception condition is raised: *invalid cursor name*.

3) If the prepared statement associated with the <dynamic cursor name> contains <dynamic parameter specification>s and an <input using clause> is not specified, then an exception condition is raised: *dynamic SQL error — using clause required for dynamic parameters*.

4) The cursor specified by <dynamic cursor name> is updatable if and only if the associated <cursor specification> specified an updatable cursor.

   NOTE 440 — "updatable cursor" is defined in Subclause 14.1, "<declare cursor>".

5) If an <input using clause> is specified, then the General Rules specified in Subclause 19.10, "<input using clause>", for <dynamic open statement> are applied.

6) All General Rules of Subclause 14.2, "<open statement>", apply to the <dynamic open statement>.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic open statement>.

19.17 <dynamic fetch statement>

Function

Fetch a row for a cursor declared with a <dynamic declare cursor>.

Format

```
<dynamic fetch statement> ::=
    FETCH [ [ <fetch orientation> ] FROM ] <dynamic cursor name> <output using clause>
```

Syntax Rules

1) If <fetch orientation> is omitted, then NEXT is implicit.

2) If <dynamic cursor name> *DCN* is a <cursor name> *CN*, then the containing <SQL-client module definition> shall contain a <dynamic declare cursor> whose <cursor name> is *CN*.

3) Let *CR* be the cursor identified by *DCN*.

4) If the implicit or explicit <fetch orientation> is not NEXT, then the <dynamic declare cursor> or <allocate cursor statement> associated with *CR* shall specify SCROLL.

Access Rules

None.

General Rules

1) All General Rules of Subclause 14.3, "<fetch statement>", are applied to cursor *CR*, <fetch orientation>, and an empty <fetch target list>.

2) The General Rules of Subclause 19.11, "<output using clause>", are applied to the <dynamic fetch statement> and the current row of *CR* as the retrieved row.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic fetch statement>.

19.18 <dynamic single row select statement>

Function

Retrieve values from a dynamically-specified row of a table.

Format

```
<dynamic single row select statement> ::= <query specification>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *Q* be the result of the <query specification>.

2) Case:

   a) If the cardinality of *Q* is greater than 1 (one), then an exception condition is raised: *cardinality violation.*

   b) If *Q* is empty, then a completion condition is raised: *no data.*

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic single row select statement>.

19.19 <dynamic close statement>

Function

Close a cursor.

Format

```
<dynamic close statement> ::= CLOSE <dynamic cursor name>
```

Syntax Rules

1) If <dynamic cursor name> *DCN* is a <cursor name> *CN*, then the containing <SQL-client module definition> shall contain a <dynamic declare cursor> whose <cursor name> is *CN*.

Access Rules

None.

General Rules

1) All General Rules of Subclause 14.4, "<close statement>", apply to the <dynamic close statement>.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic close statement>.

19.20 <dynamic delete statement: positioned>

Function

Delete a row of a table.

Format

```
<dynamic delete statement: positioned> ::=
    DELETE FROM <target table> WHERE CURRENT OF <dynamic cursor name>
```

Syntax Rules

1) If <dynamic cursor name> *DCN* is a <cursor name> *CN*, then the containing <SQL-client module definition> shall contain a <dynamic declare cursor> whose <cursor name> is *CN*.

Access Rules

1) All Access Rules of Subclause 14.6, "<delete statement: positioned>", apply to the <dynamic delete statement: positioned>.

General Rules

1) If *DCN* is a <cursor name> and the <statement name> of the associated <dynamic declare cursor> is not associated with a prepared statement, then an exception condition is raised: *invalid SQL statement name*.

2) If *DCN* is an <extended cursor name> whose value does not identify a cursor allocated in the scope of the <extended cursor name>, then an exception condition is raised: *invalid cursor name*.

3) Let *CR* be the cursor identified by *DCN*.

4) If *CR* is not an updatable cursor, then an exception condition is raised: *invalid cursor name*.

5) Let *T* be the simply underlying table of *CR*. *T* is the subject table of the <dynamic delete statement: positioned>. *T* shall have exactly one leaf underlying table *LUT*. Let *LUTN* be a <table name> that identifies *LUT*.

6) Let *TN* be the <table name> contained in <target table>. If *TN* does not identify *LUTN*, or if ONLY is specified and the <table reference> in *T* that references *LUT* does not specify ONLY, or if ONLY is not specified and the <table reference> in *T* that references *LUT* does specify ONLY, then an exception condition is raised: *target table disagrees with cursor specification*.

7) All General Rules of Subclause 14.6, "<delete statement: positioned>", apply to the <dynamic delete statement: positioned>, replacing "<delete statement: positioned>" with "<dynamic delete statement: positioned>".

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic delete statement: positioned>.

19.21 <dynamic update statement: positioned>

Function

Update a row of a table.

Format

```
<dynamic update statement: positioned> ::=
    UPDATE <target table> SET <set clause list>
    WHERE CURRENT OF <dynamic cursor name>
```

Syntax Rules

1) If <dynamic cursor name> *DCN* is a <cursor name> *CN*, then the containing <SQL-client module definition> shall contain a <dynamic declare cursor> whose <cursor name> is *CN*.

2) The scope of the <table name> is the entire <dynamic update statement: positioned>.

Access Rules

1) All Access Rules of Subclause 14.10, "<update statement: positioned>", apply to the <dynamic update statement: positioned>.

General Rules

1) If *DCN* is a <cursor name> and the <statement name> of the associated <dynamic declare cursor> is not associated with a prepared statement, then an exception condition is raised: *invalid SQL statement name*.

2) If *DCN* is an <extended cursor name> whose value does not identify a cursor allocated in the scope of the <extended cursor name>, then an exception condition is raised: *invalid cursor name*.

3) Let *CR* be the cursor identified by *DCN*.

4) If *CR* is not an updatable cursor, then an exception condition is raised: *invalid cursor name*.

5) Let *T* be the simply underlying table of *CR*. *T* is the subject table of the <dynamic update statement: positioned>. *T* shall have exactly one leaf underlying table *LUT*. Let *LUTN* be a <table name> that identifies *LUT*.

6) Let *TN* be the <table name> contained in <target table>. If *TN* does not identify *LUTN*, or if ONLY is specified and the <table reference> in *T* that references *LUT* does not specify ONLY, or if ONLY is not specified and the <table reference> in *T* that references *LUT* does specify ONLY, then an exception condition is raised: *target table disagrees with cursor specification*.

7) If any object column is directly or indirectly referenced in the <order by clause> of the <cursor specification> for *CR*, then an exception condition is raised: *attempt to assign to ordering column*.

8) If any object column identifies a column that is not identified by a <column name> contained in the explicit or implicit <column name list> of the explicit or implicit <updatability clause> of the <cursor specification> for *CR*, then an exception condition is raised: *attempt to assign to non-updatable column.*

9) All General Rules of Subclause 14.10, "<update statement: positioned>", apply to the <dynamic update statement: positioned>, replacing "<cursor name>" with "<dynamic cursor name>" and "<update statement: positioned>" with "<dynamic update statement: positioned>".

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic update statement: positioned>.

19.22 <preparable dynamic delete statement: positioned>

Function

Delete a row of a table through a dynamic cursor.

Format

```
<preparable dynamic delete statement: positioned> ::=
    DELETE [ FROM <target table> ]
    WHERE CURRENT OF [ <scope option> ] <cursor name>
```

Syntax Rules

1) If <target table> is not specified, then let *TN* be the name of the leaf underlying table *LUT* of the <cursor specification> identified by <cursor name>.

   Case:

   a) If the <table reference> that references *LUT* specifies ONLY, then the <target table>

      ONLY (*TN*)

      is implicit.

   b) Otherwise, the <target table>

      TN

      is implicit.

2) All Syntax Rules of Subclause 14.6, "<delete statement: positioned>", apply to the <preparable dynamic delete statement: positioned>, replacing "<declare cursor>" with "<dynamic declare cursor> or <allocate cursor statement>" and "<delete statement: positioned>" with "<preparable dynamic delete statement: positioned>".

Access Rules

1) All Access Rules of Subclause 14.6, "<delete statement: positioned>", apply to the <preparable dynamic delete statement: positioned>.

General Rules

1) All General Rules of Subclause 14.6, "<delete statement: positioned>", apply to the <preparable dynamic delete statement: positioned>, replacing "<delete statement: positioned>" with "<preparable dynamic delete statement: positioned>".

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <preparable dynamic delete statement: positioned>.

19.23 <preparable dynamic update statement: positioned>

Function

Update a row of a table through a dynamic cursor.

Format

```
<preparable dynamic update statement: positioned> ::=
    UPDATE [ <target table> ] SET <set clause list>
    WHERE CURRENT OF [ <scope option> ] <cursor name>
```

Syntax Rules

1) If <target table> is not specified, then let *TN* be the name of the leaf underlying table *LUT* of the <cursor specification> identified by <cursor name>.

   Case:

   a) If the <table reference> that references *LUT* specifies ONLY, then the <target table>

      ```
      ONLY ( TN )
      ```

      is implicit.

   b) Otherwise, the <target table>

      TN

      is implicit.

2) All Syntax Rules of Subclause 14.10, "<update statement: positioned>", apply to the <preparable dynamic update statement: positioned>, replacing "<declare cursor>" with "<dynamic declare cursor> or <allocate cursor statement>" and "<update statement: positioned>" with "<preparable dynamic update statement: positioned>".

Access Rules

1) All Access Rules of Subclause 14.10, "<update statement: positioned>", apply to the <preparable dynamic update statement: positioned>.

General Rules

1) All General Rules of Subclause 14.10, "<update statement: positioned>", apply to the <preparable dynamic update statement: positioned>, replacing "<update statement: positioned>" with "<preparable dynamic update statement: positioned>".

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <preparable dynamic update statement: positioned>.

This page intentionally left blank.

20 Embedded SQL

20.1 <embedded SQL host program>

Function

Specify an <embedded SQL host program>.

Format

```
<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>
  | <embedded SQL PL/I program>

<embedded SQL statement> ::=
    <SQL prefix> <statement or declaration> [ <SQL terminator> ]

<statement or declaration> ::=
    <declare cursor>
  | <dynamic declare cursor>
  | <temporary table declaration>
  | <embedded authorization declaration>
  | <embedded path specification>
  | <embedded transform group specification>
  | <embedded collation specification>
  | <embedded exception declaration>
  | <SQL procedure statement>

<SQL prefix> ::=
    EXEC SQL
  | <ampersand>SQL<left paren>

<SQL terminator> ::=
    END-EXEC
  | <semicolon>
  | <right paren>

<embedded authorization declaration> ::= DECLARE <embedded authorization clause>

<embedded authorization clause> ::=
    SCHEMA <schema name>
  | AUTHORIZATION <embedded authorization identifier>
    [ FOR STATIC { ONLY | AND DYNAMIC } ]
```

```
  | SCHEMA <schema name> AUTHORIZATION <embedded authorization identifier>
    [ FOR STATIC { ONLY | AND DYNAMIC } ]

<embedded authorization identifier> ::=
    <module authorization identifier>

<embedded path specification> ::= <path specification>

<embedded transform group specification> ::=
    <transform group specification>

<embedded collation specification> ::= <module collations>

<embedded SQL declare section> ::=
    <embedded SQL begin declare>
    [ <embedded character set declaration> ]
    [ <host variable definition>... ]
    <embedded SQL end declare>
  | <embedded SQL MUMPS declare>

<embedded character set declaration> ::=
    SQL NAMES ARE <character set specification>

<embedded SQL begin declare> ::=
    <SQL prefix> BEGIN DECLARE SECTION [ <SQL terminator> ]

<embedded SQL end declare> ::=
    <SQL prefix> END DECLARE SECTION [ <SQL terminator> ]

<embedded SQL MUMPS declare> ::=
    <SQL prefix>
    BEGIN DECLARE SECTION
    [ <embedded character set declaration> ]
    [ <host variable definition>... ]
    END DECLARE SECTION
    <SQL terminator>

<host variable definition> ::=
    <Ada variable definition>
  | <C variable definition>
  | <COBOL variable definition>
  | <Fortran variable definition>
  | <MUMPS variable definition>
  | <Pascal variable definition>
  | <PL/I variable definition>

<embedded variable name> ::= <colon><host identifier>

<host identifier> ::=
    <Ada host identifier>
  | <C host identifier>
  | <COBOL host identifier>
  | <Fortran host identifier>
  | <MUMPS host identifier>
  | <Pascal host identifier>
  | <PL/I host identifier>
```

Syntax Rules

1) An <embedded SQL host 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.

   NOTE 441 — "Compilation unit" is defined in Subclause 4.22, "SQL-client modules".

2) Case:

   a) An <embedded SQL statement> or <embedded SQL MUMPS declare> that is contained in an <embedded SQL MUMPS program> shall contain an <SQL prefix> that is "<ampersand>SQL<left paren>". There shall be no <separator> between the <ampersand> and "SQL" nor between "SQL" and the <left paren>.

   b) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> that is not contained in an <embedded SQL MUMPS program> shall contain an <SQL prefix> that is "EXEC SQL".

3) Case:

   a) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> contained in an <embedded SQL COBOL program> shall contain an <SQL terminator> that is END-EXEC.

   b) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> contained in an <embedded SQL Fortran program> shall not contain an <SQL terminator>.

   c) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> contained in an <embedded SQL Ada program>, <embedded SQL C program>, <embedded SQL Pascal program>, or <embedded SQL PL/I program> shall contain an <SQL terminator> that is a <semicolon>.

   d) An <embedded SQL statement> or <embedded SQL MUMPS declare> that is contained in an <embedded SQL MUMPS program> shall contain an <SQL terminator> that is a <right paren>.

4) Case:

   a) An <embedded SQL declare section> that is contained in an <embedded SQL MUMPS program> shall be an <embedded SQL MUMPS declare>.

   b) An <embedded SQL declare section> that is not contained in an <embedded SQL MUMPS program> shall not be an <embedded SQL MUMPS declare>.

   NOTE 442 — There is no restriction on the number of <embedded SQL declare section>s that may be contained in an <embedded SQL host program>.

5) The <token>s comprising an <SQL prefix>, <embedded SQL begin declare>, or <embedded SQL end declare> shall be separated by <space> characters and shall be specified on one line. Otherwise, the rules for the continuation of lines and tokens from one line to the next and for the placement of host language comments are those of the programming language of the containing <embedded SQL host program>.

6) If an <embedded authorization declaration> appears in an <embedded SQL host program>, then it shall be contained in the first <embedded SQL statement> of that <embedded SQL host program>.

7) An <embedded SQL host program> shall not contain more than one <embedded path specification>.

8) An <embedded SQL host program> shall not contain more than one <embedded transform group specification>.

9) An <embedded SQL host program> shall not contain more than one <embedded collation specification>.

10) Case:

   a) If <embedded transform group specification> is not specified, then an <embedded transform group specification> containing a <multiple group specification> with a <group specification> *GS* for each <host variable definition> that has an associated user-defined type *UDT*, but is not a user-defined locator variable is implicit. The <group name> of *GS* is implementation-defined and its <path-resolved user-defined type name> is the <user-defined type name> of *UDT*.

   b) If <embedded transform group specification> contains a <single group specification> with a <group name> *GN*, then an <embedded transform group specification> containing a <multiple group specification> with a <group specification> *GS* for each <host variable definition> that has an associated user-defined type *UDT*, but is not a user-defined type locator variable is implicit. The <group name> of *GS* is *GN* and its <path-resolved user-defined type name> is the <user-defined type name> of *UDT*.

   c) If <embedded transform group specification> contains a <multiple group specification> *MGS*, then an <embedded transform group specification> containing a <multiple group specification> that contains *MGS* extended with a <group specification> *GS* for each <host variable definition> that has an associated user-defined type *UDT*, but is not a user-defined locator variable and no equivalent of *UDT* is contained in any <group specification> contained in *MGS* is implicit. The <group name> of *GS* is implementation-defined and its <path-resolved user-defined type name> is the <user-defined type name> of *UDT*.

11) In the text of the <embedded SQL host program>, the implicit or explicit <embedded transform group specification> shall precede every <host variable definition>.

12) An <embedded SQL host program> shall contain no more than one <embedded character set declaration>. If an <embedded character set declaration> is not specified, then an <embedded character set declaration> that specifies an implementation-defined character set that contains at least every character that is in <SQL language character> is implicit.

13) A <temporary table declaration> that is contained in an <embedded SQL host program> shall precede in the text of that <embedded SQL host program> any SQL-statement or <declare cursor> that references the <table name> of the <temporary table declaration>.

14) A <declare cursor> that is contained in an <embedded SQL host program> shall precede in the text of that <embedded SQL host program> any SQL-statement that references the <cursor name> of the <declare cursor>.

15) A <dynamic declare cursor> that is contained in an <embedded SQL host program> shall precede in the text of that <embedded SQL host program> any SQL-statement that references the <cursor name> of the <dynamic declare cursor>.

16) Any <host identifier> that is contained in an <embedded SQL statement> in an <embedded SQL host program> shall be defined in exactly one <host variable definition> contained in that <embedded SQL host program>. In programming languages that support <host variable definition>s in subprograms, two <host variable definition>s with different, non-overlapping scope in the host language are to be regarded as defining different host variables, even if they specify the same variable name. That <host variable definition> shall appear in the text of the <embedded SQL host program> prior to any <embedded SQL statement>

that references the <host identifier>. The <host variable definition> shall be such that a host language reference to the <host identifier> is valid at every <embedded SQL statement> that contains the <host identifier>.

17) A <host variable definition> defines the host language data type of the <host identifier>. For every such host language data type an equivalent SQL <data type> is specified in Subclause 20.3, "<embedded SQL Ada program>", Subclause 20.4, "<embedded SQL C program>", Subclause 20.5, "<embedded SQL COBOL program>", Subclause 20.6, "<embedded SQL Fortran program>", Subclause 20.7, "<embedded SQL MUMPS program>", Subclause 20.8, "<embedded SQL Pascal program>", and Subclause 20.9, "<embedded SQL PL/I program>".

18) An <embedded SQL host program> shall contain a <host variable definition> that specifies SQLSTATE.

19) If one or more <host variable definition>s that specify SQLSTATE appear in an <embedded SQL host program>, then the <host variable definition>s shall be such that a host language reference to SQLSTATE is valid at every <embedded SQL statement>, including <embedded SQL statement>s that appear in any subprograms contained in that <embedded SQL host program>. The first such <host variable definition> of SQLSTATE shall appear in the text of the <embedded SQL host program> prior to any <embedded SQL statement>.

20) Given an <embedded SQL host program> *H*, there is an implied standard-conforming <SQL-client module definition> *M* and an implied standard-conforming host program *P* derived from *H*. The derivation of the implied program *P* and the implied <SQL-client module definition> *M* of an <embedded SQL host program> *H* effectively precedes the processing of any host language program text manipulation commands such as inclusion or copying of text.

NOTE 443 — Before *H* can be executed, *M* is processed by an implementation-defined mechanism to produce an SQL-client module. An SQL-implementation may combine this mechanism with the processing of the <embedded SQL host program>, in which the existence of *M* is pure hypothetical.

Given an <embedded SQL host program> *H* with an implied <SQL-client module definition> *M* and an implied program *P* defined as above:

a) The implied <SQL-client module definition> *M* of *H* shall be a standard-conforming <SQL-client module definition>.

b) If *H* is an <embedded SQL Ada program>, an <embedded SQL C program>, an <embedded SQL COBOL program>, an <embedded SQL Fortran program>, an <embedded SQL MUMPS program>, an <embedded SQL Pascal program>, or an <embedded SQL PL/I program>, then the implied program *P* shall be a standard-conforming Ada program, a standard-conforming C program, a standard-conforming COBOL program, a standard-conforming Fortran program, a standard-conforming M program, a standard-conforming Pascal program, or standard-conforming PL/I program, respectively.

21) *M* is derived from *H* as follows:

a) *M* contains a <module name clause> whose <SQL-client module name> is either implementation-dependent or is omitted.

b) *M* contains a <module character set specification> that is identical to the explicit or implicit <embedded character set declaration> with the keyword "SQL" removed.

c) *M* contains a <language clause> that specifies either ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, where *H* is respectively an <embedded SQL Ada program>, an <embedded SQL C program>, an

<embedded SQL COBOL program>, an <embedded SQL Fortran program>, an <embedded SQL MUMPS program>, an <embedded SQL Pascal program>, or an <embedded SQL PL/I program>.

d) Case:

   i) If *H* contains an <embedded authorization declaration> *EAD*, then let *EAC* be the <embedded authorization clause>contained in *EAD*; *M* contains a <module authorization clause> that specifies *EAC*.

   ii) Otherwise, let *SN* be an implementation-defined <schema name>; *M* contains a <module authorization clause> that specifies "SCHEMA *SN*".

e) Case:

   i) If *H* contains an <embedded path specification> *EPS*, then *M* contains the <module path specification> *EPS*.

   ii) Otherwise, *M* contains an implementation-defined <module path specification>.

f) *M* contains a <module transform group specification> that is identical to the explicit or implicit <embedded transform group specification>.

g) If an <embedded collation specification> *ECS* is specified, then *M* contains a <module collations> that is identical to the <module collations> contained in *ECS*.

h) For every <declare cursor> *EC* contained in *H*, *M* contains one <declare cursor> *PC* and one <externally-invoked procedure> *PS* that contains an <open statement> that references *PC*.

   i) The <procedure name> of *PS* is implementation-dependent. *PS* contains a <host parameter declaration> *PD* for each distinct <embedded variable name> *EVN* contained in *PC* with an implementation-dependent <host parameter name> *PN* and the <host parameter data type> *PT*, determined as follows:

      Case:

      1) If *EVN* identifies a binary large object locator variable, then *PT* is BLOB AS LOCATOR.

      2) If *EVN* identifies a character large object locator variable, then *PT* is CLOB AS LOCATOR.

      3) If *EVN* identifies an array locator variable, then *PT* is *AAT* AS LOCATOR, where *AAT* is the associated array type of *V*.

      4) If *EVN* identifies a multiset locator variable, then *PT* is *AMT* AS LOCATOR, where *AMT* is the associated multiset type of *V*.

      5) If *EVN* identifies a user-defined type locator variable, then *PT* is *UDT* AS LOCATOR, where *UDT* is the associated user-defined type of *V*.

      6) Otherwise, *PT* is the SQL data type that corresponds to the host language data type of *EVN* as specified in Subclause 13.6, "Data type correspondences".

   ii) *PS* contains a <host parameter declaration> that specifies SQLSTATE. The order of <host parameter declaration>s in *PS* is implementation-dependent. *PC* is a copy of *EC* in which each *EVN* has been replaced as follows:

      Case:

1) If *EVN* does not identify user-defined type locator variable, but *EVN* identifies a host variable that has an associated user-defined type *UT*, then:

   A) Let *GN* be the <group name> corresponding to the <user-defined type name> of *UT* contained in <group specification> contained in <embedded transform group specification>.

   B) Apply the Syntax Rules of Subclause 9.19, "Determination of a to-sql function", with *DT* and *GN* as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function *TSF*.

   C) Let the declared type of the single SQL parameter of *TSF* be *TPT*. *PT* shall be assignable to *TPT*.

   D) *EVN* is replaced by:

```
TSFN(CAST(PN AS TPT))
```

2) Otherwise, *EVN* is replaced by:

```
PN
```

i) For every <dynamic declare cursor> *EC* in *H*, *M* contains one <dynamic declare cursor> *PC* that is a copy of *EC*.

j) *M* contains one <temporary table declaration> for each <temporary table declaration> contained in *H*. Each <temporary table declaration> of *M* is a copy of the corresponding <temporary table declaration> of *H*.

k) *M* contains one <embedded exception declaration> for each <embedded exception declaration> contained in *H*. Each <embedded exception declaration> of *M* is a copy of the corresponding <embedded exception declaration> of *H*.

l) *M* contains an <externally-invoked procedure> for each <SQL procedure statement> contained in *H*. The <externally-invoked procedure> *PS* of *M* corresponding with an <SQL procedure statement> *ES* of *H* is defined as follows.

   Case:

   i) If *ES* is not an <open statement>, then:

      1) The <procedure name> of *PS* is implementation-dependent.

      2) Let *n* be the number of distinct <embedded variable name>s contained in *ES*. Let HVN_i, 1 (one) $\le i \le n$, be the *i*-th such <embedded variable name> and let HV_i be the host variable identified by HVN_i.

      3) For each HVN_i, 1 (one) $\le i \le n$, *PS* contains a <host parameter declaration> PD_i defining a host parameter P_i such that:

         A) The <host parameter name> PN_i of PD_i is implementation-dependent.

         B) The <host parameter data type> PT_i of PD_i is determined as follows.

Case:

I) If HV_i is a binary large object locator variable, then PT_i is BLOB AS LOCATOR.

II) If HV_i is a character large object locator variable, then PT_i is CLOB AS LOCATOR.

III) If HV_i is an array locator variable, then PT_i is *AAT* AS LOCATOR, where *AAT* is the associated array type of HV_i.

IV) If HV_i is a multiset locator variable, then PT_i is *AMT* AS LOCATOR, where *AMT* is the associated multiset type of HV_i.

V) If HV_i is user-defined type locator variable, then PT_i is *UDT* AS LOCATOR, where *UDT* is the associated user-defined type of HV_i.

VI) Otherwise, PT_i is the SQL data type that corresponds to the host language data type of HV_i as specified in Subclause 13.6, "Data type correspondences".

4) *PS* contains a <host parameter declaration> that specifies SQLSTATE.

5) The order of the <host parameter declaration>s PD_i, 1 (one) $\le i \le n$, is implementation-dependent.

6) For each HVN_i, 1 (one) $\le i \le n$, that identifies some HV_i that has an associated user-defined type, but is not a user-defined type locator variable, apply the Syntax Rules of Subclause 9.6, "Host parameter mode determination", with the PD_i corresponding to HVN_i and *ES* as <host parameter declaration> and <SQL procedure statement>, respectively, to determine whether the corresponding P_i is an input host parameter, an output host parameter, or both an input host parameter and an output host parameter.

A) Among P_i, 1 (one) $\le i \le n$, let a be the number of input host parameters, b be the number of output host parameters, and let c be the number of host parameters that are both input host parameters and output host parameters.

B) Among P_i, 1 (one) $\le i \le n$, let PI_j, 1 (one) $\le j \le a$, be the input host parameters, let PO_k, 1 (one) $\le k \le b$, be the output host parameters, and let PIO_l, 1 (one) $\le l \le c$, be the host parameters that are both input host parameters and output host parameters.

C) Let PNI_j, 1 (one) $\le j \le a$, be the <host parameter name> of PI_j. Let PNO_k, 1 (one) $\le k \le b$, be the <host parameter name> of PO_k. Let $PNIO_l$, 1 (one) $\le l \le c$, be the <host parameter name> of PIO_l.

D) Let HVI_j, 1 (one) $\le j \le a$, be the host variable corresponding to PI_j. Let HVO_k, 1 (one) $\le k \le b$, be the host variable corresponding to PO_k. Let $HVIO_l$, 1 (one) $\le l \le c$, be the host variable corresponding to PIO_l.

E) Let TSI_j, 1 (one) $\le j \le a$, be the associated SQL data type of HVI_j. Let TSO_k, 1 (one) $\le k \le b$, be the associated SQL data type of HVO_k. Let $TSIO_l$, 1 (one) $\le l \le c$, be the associated SQL data type of $HVIO_l$.

F) Let TUI_j, 1 (one) $\le j \le a$, be the associated user-defined type of HVI_j. Let TUO_k, 1 (one) $\le k \le b$, be the associated user-defined type of HVO_k. Let $TUIO_l$, 1 (one) $\le l \le c$, be the associated user-defined type of $HVIO_l$.

G) Let GNI_j, 1 (one) $\le j \le a$, be the <group name> corresponding to the <user-defined type name> of TUI_j contained in the <group specification> contained in <embedded transform group specification>. Let GNO_k, 1 (one) $\le k \le b$, be the <group name> corresponding to the <user-defined type name> of TUO_k contained in the <group specification> contained in <embedded transform group specification>. Let $GNIO_l$, 1 (one) $\le l \le c$, be the <group name> corresponding to the <user-defined type name> of $TUIO_l$ contained in the <group specification> contained in <embedded transform group specification>.

H) For every j, 1 (one) $\le j \le a$, apply the Syntax Rules of Subclause 9.19, "Determination of a to-sql function", with TUI_j and GNI_j as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function $TSFI_j$ identified by <routine name> $TSIN_j$. Let TTI_j be the data type of the single SQL parameter of $TSFI_j$. TSI_j shall be assignable to TTI_j.

I) For every l, 1 (one) $\le l \le c$, apply the Syntax Rules of Subclause 9.19, "Determination of a to-sql function", with $TUIO_l$ and $GNIO_l$ as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function $TSFIO_l$ identified by <routine name> $TSION_l$. Let $TTIO_l$ be the data type of the single SQL parameter of $TSFIO_l$. $TSIO_l$ shall be assignable to $TTIO_l$.

J) For every k, 1 (one) $\le k \le b$, apply the Syntax Rules of Subclause 9.17, "Determination of a from-sql function", with TUO_k and GNO_k as *TYPE* and *GROUP*, respectively. There shall be an applicable from-sql function $FSFO_k$ identified by <routine name> $FSON_k$. Let TRO_k be the result data type of $FSFO_k$. TSO_k shall be assignable to TRO_k.

K) For every l, 1 (one) $\le l \le c$, apply the Syntax Rules of Subclause 9.17, "Determination of a from-sql function", with $TUIO_l$ and $GNIO_l$ as *TYPE* and *GROUP*, respectively. There shall be an applicable from-sql function $FSFIO_l$ identified by <routine name> $FSION_l$. Let $TRIO_l$ be the result data type of $FSFIO_l$. $TSIO_l$ shall be assignable to $TRIO_l$.

L) Let SVI_j, 1 (one) $\le j \le a$, SVO_k, 1 (one) $\le k \le b$, and $SVIO_l$, 1 (one) $\le l \le c$, be implementation-dependent <SQL variable name>s, each of which is not equivalent to any other <SQL variable name> contained in *ES*, to any <SQL parameter name> contained in *ES*, or to any <column name> contained in *ES*.

7) Let *NES* be an <SQL procedure statement> that is a copy of *ES* in which every HVN_i, 1 (one) $\leq i \leq n$, is replaced as follows.

Case:

A) If HV_i has an associated user-defined type but is not a user-defined type locator variable, then

Case:

I) If P_i is an input host parameter, then let PI_j, 1 (one) $\leq j \leq a$, be the input host parameter that corresponds to P_i; HVN_i is replaced by SVI_j.

II) If P_i is an output host parameter, then let PO_k, 1 (one) $\leq k \leq b$, be the output host parameter that corresponds to P_i; HVN_i is replaced by SVO_k.

III) Otherwise, let PIO_l, 1 (one) $\leq l \leq c$, be the input host parameter and the output host parameter that corresponds to P_i; HVN_i is replaced by $SVIO_l$.

B) Otherwise, HVN_i is replaced by PN_i.

8) The <SQL procedure statement> of *PS* is:

```
BEGIN ATOMIC
    DECLARE SVI1 TUI1;
    ...
    DECLARE SVIa TUIa;
    DECLARE SVO1 TUO1;
    ...
    DECLARE SVOb TUOb;
    DECLARE SVIO1 TUIO1;
    ...
    DECLARE SVIOc TUIOc;
    SET SVI1 = TSIN1 (CAST (PNI1 AS TTI1));
    ...
    SET SVIa = TSINa (CAST (PNIa AS TTIa));
    SET SVIO1 = TSION1 (CAST (PNIO1 AS TTIO1));
    ...
    SET SVIOc = TSIONc (CAST (PNIOc AS TTIOc));
    NES;
    SET PNO1 = CAST ( FSON1 (SVO1) AS TSO1);
    ...
    SET PNOb = CAST ( FSONb (SVOb) AS TSOb);
    SET PNIO1 = CAST ( FSION1 (SVIO1) AS TSIO1);
    ...
    SET PNIOc = CAST ( FSIONc (SVIOc) AS TSIOc);
END;
```

9) Whether one <externally-invoked procedure> of *M* can correspond to more than one <SQL procedure statement> of *H* is implementation-dependent.

ii) If *ES* is an <open statement>, then:

1) Let *EC* be the <declare cursor> in *H* referenced by *ES*.

2) *PS* is the <externally-invoked procedure> in *M* that contains an <open statement> that references the <declare cursor> in *M* corresponding to *EC*.

22) *P* is derived from *H* as follows:

a) Each <embedded SQL begin declare>, <embedded SQL end declare>, and <embedded character set declaration> has been deleted. If the embedded host language is M, then each <embedded SQL MUMPS declare> has been deleted.

b) Each <host variable definition> in an <embedded SQL declare section> has been replaced by a valid data definition in the target host language according to the Syntax Rules specified in an <embedded SQL Ada program>, <embedded SQL C program>, <embedded SQL COBOL program>, <embedded SQL Fortran program>, <embedded SQL Pascal program>, or an <embedded SQL PL/I program> clause.

c) Each <embedded SQL statement> that contains a <declare cursor>, a <dynamic declare cursor>, an <SQL-invoked routine>, or a <temporary table declaration> has been deleted, and every <embedded SQL statement> that contains an <embedded exception declaration> has been replaced with statements of the host language that will have the effect specified by the General Rules of Subclause 20.2, "<embedded exception declaration>".

d) Each <embedded SQL statement> that contains an <SQL procedure statement> has been replaced by host language statements that perform the following actions:

i) A host language procedure or subroutine call of the <externally-invoked procedure> of the implied <SQL-client module definition> *M* of *H* that corresponds with the <SQL procedure statement>.

If the <SQL procedure statement> is not an <open statement>, then the arguments of the call include each distinct <host identifier> contained in the <SQL procedure statement> together with the SQLSTATE <host identifier>. If the <SQL procedure statement> is an <open statement>, then the arguments of the call include each distinct <host identifier> contained in the corresponding <declare cursor> of *H* together with the SQLSTATE <host identifier>.

The order of the arguments in the call corresponds with the order of the corresponding <host parameter declaration>s in the corresponding <externally-invoked procedure>.

NOTE 444 — In an <embedded SQL Fortran program>, the "SQLSTATE" variable may be abbreviated to "SQLSTA". See the Syntax Rules of Subclause 20.6, "<embedded SQL Fortran program>".

ii) Exception actions, as specified in Subclause 20.2, "<embedded exception declaration>".

e) Each <statement or declaration> that contains an <embedded authorization declaration> is deleted.

Access Rules

1) For every host variable whose <embedded variable name> is contained in <statement or declaration> and has an associated user-defined type, the current privileges shall include EXECUTE privilege on all from-sql functions (if any) and all to-sql functions (if any) referenced in the corresponding SQL-client module.

General Rules

1) The interpretation of an <embedded SQL host program> *H* is defined to be equivalent to the interpretation of the implied program *P* of *H* and the implied <SQL-client module definition> *M* of *H*.

Conformance Rules

1) Without Feature B051, "Enhanced execution rights", conforming SQL language shall not contain an <embedded authorization declaration>.

2) Without Feature F461, "Named character sets", conforming SQL language shall not contain an <embedded character set declaration>.

3) Without Feature F361, "Subprogram support", conforming SQL language shall not contain two <host variable definition>s that specify the same variable name.

4) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain an <embedded path specification>.

5) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <embedded transform group specification>.

20.2 <embedded exception declaration>

Function

Specify the action to be taken when an SQL-statement causes a specific class of condition to be raised.

Format

```
<embedded exception declaration> ::= WHENEVER <condition> <condition action>

<condition> ::= <SQL condition>

<SQL condition> ::=
    <major category>
  | SQLSTATE ( <SQLSTATE class value> [ , <SQLSTATE subclass value> ] )
  | CONSTRAINT <constraint name>

<major category> ::=
    SQLEXCEPTION
  | SQLWARNING
  | NOT FOUND

<SQLSTATE class value> ::=
    <SQLSTATE char><SQLSTATE char> !! See the Syntax Rules.

<SQLSTATE subclass value> ::=
    <SQLSTATE char><SQLSTATE char><SQLSTATE char> !! See the Syntax Rules.

<SQLSTATE char> ::=
    <simple Latin upper case letter>
  | <digit>

<condition action> ::=
    CONTINUE
  | <go to>

<go to> ::= { GOTO | GO TO } <goto target>

<goto target> ::=
    <host label identifier>
  | <unsigned integer>
  | <host PL/I label variable>

<host label identifier> ::= !! See the Syntax Rules.

<host PL/I label variable> ::= !! See the Syntax Rules.
```

Syntax Rules

1) SQLWARNING, NOT FOUND, and SQLEXCEPTION correspond to SQLSTATE class values corresponding to categories W, N, and X in Table 32, "SQLSTATE class and subclass values", respectively.

2) An <embedded exception declaration> contained in an <embedded SQL host program> applies to an <SQL procedure statement> contained in that <embedded SQL host program> if and only if the <SQL procedure statement> appears after the <embedded exception declaration> that has condition *C* in the text sequence of the <embedded SQL host program> and no other <embedded exception declaration> *E* that satisfies one of the following conditions appears between the <embedded exception declaration> and the <SQL procedure statement> in the text sequence of the <embedded SQL host program>.

   Let *D* be the <condition> contained in *E*.

   a) *D* is the same as *C*.

   b) *D* is a <major category> and belongs to the same class to which *C* belongs.

   c) *D* contains an <SQLSTATE class value>, but does not contain an <SQLSTATE subclass value>, and *E* contains the same <SQLSTATE class value> that *C* contains.

   d) *D* contains the <SQLSTATE class value> that corresponds to *integrity constraint violation* and *C* contains CONSTRAINT.

3) In the values of <SQLSTATE class value> and <SQLSTATE subclass value>, there shall be no <separator> between the <SQLSTATE char>s.

4) The values of <SQLSTATE class value> and <SQLSTATE subclass value> shall correspond to class values and subclass values, respectively, specified in Table 32, "SQLSTATE class and subclass values".

5) If an <embedded exception declaration> specifies a <go to>, then the <host label identifier>, <host PL/I label variable>, or <unsigned integer> of the <go to> shall be such that a host language GO TO statement specifying that <host label identifier>, <host PL/I label variable>, or <unsigned integer> is valid at every <SQL procedure statement> to which the <embedded exception declaration> applies.

   NOTE 445 —

   If an <embedded exception declaration> is contained in an <embedded SQL Ada program>, then the <goto target> of a <go to> should specify a <host label identifier> that is a label_name in the containing <embedded SQL Ada program>.

   If an <embedded exception declaration> is contained in an <embedded SQL C program>, then the <goto target> of a <go to> should specify a <host label identifier> that is a label in the containing <embedded SQL C program>.

   If an <embedded exception declaration> is contained in an <embedded SQL COBOL program>, then the <goto target> of a <go to> should specify a <host label identifier> that is a section-name or an unqualified paragraph-name in the containing <embedded SQL COBOL program>.

   If an <embedded exception declaration> is contained in an <embedded SQL Fortran program>, then the <goto target> of a <go to> should be an <unsigned integer> that is the statement label of an executable statement that appears in the same program unit as the <go to>.

   If an <embedded exception declaration> is contained in an <embedded SQL MUMPS program>, then the <goto target> of a <go to> should be a gotoargument that is the statement label of an executable statement that appears in the same <embedded SQL MUMPS program>.

   If an <embedded exception declaration> is contained in an <embedded SQL Pascal program>, then the <goto target> of a <go to> should be an <unsigned integer> that is a label.

   If an <embedded exception declaration> is contained in an <embedded SQL PL/I program>, then the <goto target> of a <go to> should specify either a <host label identifier> or a <host PL/I label variable>.

   Case:

   — If <host label identifier> is specified, then the <host label identifier> should be a label constant in the containing <embedded SQL PL/I program>.

— If <host PL/I label variable> is specified, then the <host PL/I label variable> should be a PL/I label variable declared in the containing <embedded SQL PL/I program>.

Access Rules

None.

General Rules

1) Immediately after the execution of an <SQL procedure statement> *STMT* in an <embedded SQL host program> that returns an SQLSTATE value other than *successful completion*:

   a) Let *E* be the set of <embedded exception declaration>s that are contained in the <embedded SQL host program> containing *STMT*, that applies to *STMT*, and that specifies a <condition action> that is <go to>.

   b) Let *CV* and *SCV* be respectively the values of the class and subclass of the SQLSTATE value that indicates the result of the <SQL procedure statement>.

   c) If the execution of the <SQL procedure statement> caused the violation of one or more constraints or assertions, then:

      i) Let *ECN* be the set of <embedded exception declaration>s in *E* that specify CONSTRAINT and the <constraint name> of a constraint that was violated by execution of *STMT*.

      ii) If *ECN* contains more than one <embedded exception declaration>, then an implementation-dependent <embedded exception declaration> is chosen from *ECN*; otherwise, the single <embedded exception declaration> in *ECN* is chosen.

      iii) A GO TO statement of the host language is performed, specifying the <host label identifier>, <host PL/I label variable>, or <unsigned integer> of the <go to> specified in the <embedded exception declaration> chosen from *ECN*.

   d) Otherwise:

      i) Let *ECS* be the set of <embedded exception declaration>s in *E* that specify SQLSTATE, an <SQLSTATE class value>, and an <SQLSTATE subclass value>.

      ii) If *ECS* contains an <embedded exception declaration> *EY* that specifies an <SQLSTATE class value> identical to *CV* and an <SQLSTATE subclass value> identical to *SCV*, then a GO TO statement of the host language is performed, specifying the <host label identifier>, <host PL/I label variable>, or <unsigned integer> of the <go to> specified in the <embedded exception declaration> *EY*.

      iii) Otherwise:

         1) Let *EC* be the set of <embedded exception declaration>s in *E* that specify SQLSTATE and an <SQLSTATE class value> without an <SQLSTATE subclass value>.

         2) If *EC* contains an <embedded exception declaration> *EY* that specifies an <SQLSTATE class value> identical to *CV*, then a GO TO statement of the host language is performed,

specifying the <host label identifier>, <host PL/I label variable>, or <unsigned integer> of the <go to> specified in the <embedded exception declaration> *EY*.

3) Otherwise:

   A) Let *EX* be the set of <embedded exception declaration>s in *E* that specify SQLEXCEPTION.

   B) If *EX* contains an <embedded exception declaration> *EY* and *CV* belongs to Category X in Table 32, "SQLSTATE class and subclass values", then a GO TO statement of the host language is performed, specifying the <host label identifier>, <host PL/I label variable>, or <unsigned integer> of the <go to> specified in the <embedded exception declaration> *EY*.

   C) Otherwise:

      I) Let *EW* be the set of <embedded exception declaration>s in *E* that specify SQLWARNING.

      II) If *EW* contains an <embedded exception declaration> *EY* and *CV* belongs to Category W in Table 32, "SQLSTATE class and subclass values", then a GO TO statement of the host language is performed, specifying the <host label identifier>, <host PL/I label variable>, or <unsigned integer> of the <go to> specified in the <embedded exception declaration> *EY*.

      III) Otherwise, let *ENF* be the set of <embedded exception declaration>s in *E* that specify NOT FOUND. If *ENF* contains an <embedded exception declaration> *EY* and *CV* belongs to Category N in Table 32, "SQLSTATE class and subclass values", then a GO TO statement of the host language is performed, specifying the <host label identifier>, <host PL/I label variable>, or <unsigned integer> of the <go to> specified in the <embedded exception declaration> *EY*.

Conformance Rules

1) Without Feature B041, "Extensions to embedded SQL exception declarations", conforming SQL language shall not contain an <SQL condition> that contains either SQLSTATE or CONSTRAINT.

2) Without Feature F491, "Constraint management", conforming SQL language shall not contain an <SQL condition> that contains a <constraint name>.

20.3 <embedded SQL Ada program>

Function

Specify an <embedded SQL Ada program>.

Format

```
<embedded SQL Ada program> ::= !! See the Syntax Rules.

<Ada variable definition> ::=
    <Ada host identifier> [ { <comma> <Ada host identifier> }... ] <colon>
    <Ada type specification> [ <Ada initial value> ]

<Ada initial value> ::=
    <Ada assignment operator> <character representation>...

<Ada assignment operator> ::= <colon><equals operator>

<Ada host identifier> ::= !! See the Syntax Rules.

<Ada type specification> ::=
    <Ada qualified type specification>
  | <Ada unqualified type specification>
  | <Ada derived type specification>

<Ada qualified type specification> ::=
    Interfaces.SQL <period> CHAR
    [ CHARACTER SET [ IS ] <character set specification> ]
    <left paren> 1 <double period> <length> <right paren>
  | Interfaces.SQL <period> SMALLINT
  | Interfaces.SQL <period> INT
  | Interfaces.SQL <period> BIGINT
  | Interfaces.SQL <period> REAL
  | Interfaces.SQL <period> DOUBLE_PRECISION
  | Interfaces.SQL <period> BOOLEAN
  | Interfaces.SQL <period> SQLSTATE_TYPE
  | Interfaces.SQL <period> INDICATOR_TYPE

<Ada unqualified type specification> ::=
    CHAR <left paren> 1 <double period> <length> <right paren>
  | SMALLINT
  | INT
  | BIGINT
  | REAL
  | DOUBLE_PRECISION
  | BOOLEAN
  | SQLSTATE_TYPE
  | INDICATOR_TYPE

<Ada derived type specification> ::=
    <Ada CLOB variable>
  | <Ada CLOB locator variable>
  | <Ada BLOB variable>
```

```
  | <Ada BLOB locator variable>
  | <Ada user-defined type variable>
  | <Ada user-defined type locator variable>
  | <Ada REF variable>
  | <Ada array locator variable>
  | <Ada multiset locator variable>

<Ada CLOB variable> ::=
    SQL TYPE IS CLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]

<Ada CLOB locator variable> ::= SQL TYPE IS CLOB AS LOCATOR

<Ada BLOB variable> ::=
    SQL TYPE IS BLOB <left paren> <large object length> <right paren>

<Ada BLOB locator variable> ::= SQL TYPE IS BLOB AS LOCATOR

<Ada user-defined type variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS <predefined type>

<Ada user-defined type locator variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR

<Ada REF variable> ::= SQL TYPE IS <reference type>

<Ada array locator variable> ::= SQL TYPE IS <array type> AS LOCATOR

<Ada multiset locator variable> ::= SQL TYPE IS <multiset type> AS LOCATOR
```

Syntax Rules

1) An <embedded SQL Ada program> is a compilation unit that consists of Ada text and SQL text. The Ada text shall conform to [ISO8652]. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) An <embedded SQL statement> may be specified wherever an Ada statement may be specified. An <embedded SQL statement> may be prefixed by an Ada label.

3) An <Ada host identifier> is any valid Ada identifier. An <Ada host identifier> shall be contained in an <embedded SQL Ada program>.

4) An <Ada variable definition> defines one or more host variables.

5) An <Ada variable definition> shall be modified as follows before it is placed into the program derived from the <embedded SQL Ada program> (see the Syntax Rules of Subclause 20.1, "<embedded SQL host program>"):

   a) Any optional CHARACTER SET specification shall be removed from an <Ada qualified type specification> and <Ada derived type specification>.

   b) The <length> specified in a CHAR declaration of any <Ada qualified type specification> or <Ada derived type specification> that contains a CHARACTER SET specification shall be replaced by a length equal to the length in octets of *PN*, where *PN* is the <Ada host identifier> specified in the containing <Ada variable definition>.

c) The syntax

```
SQL TYPE IS CLOB ( L )
```

and the syntax

```
SQL TYPE IS BLOB ( L )
```

for a given <Ada host identifier> *HVN* shall be replaced by

```
TYPE HVN IS RECORD
  HVN_RESERVED : Interfaces.SQL.INT;
  HVN_LENGTH : Interfaces.SQL.INT;
  HVN_DATA : Interfaces.SQL.CHAR(1..L);
END RECORD;
```

in any <Ada CLOB variable> or <Ada BLOB variable>, where *L* is the numeric value of <large object length> as specified in Subclause 5.2, "<token> and <separator>".

d) The syntax

```
SQL TYPE IS UDTN AS PDT
```

shall be replaced by

```
ADT
```

in any <Ada user-defined type variable>, where *ADT* is the data type listed in the "Ada data type" column corresponding to the row for SQL data type *PDT* in Table 16, "Data type correspondences for Ada". *ADT* shall not be "none". The data type identified by *UDTN* is called the *associated user-defined type* of the host variable and the data type identified by *PDT* is called the *associated SQL data type* of the host variable.

e) The syntax

```
SQL TYPE IS BLOB AS LOCATOR
```

shall be replaced by

```
Interfaces.SQL.INT
```

in any <Ada BLOB locator variable>. The host variable defined by <Ada BLOB locator variable> is called a *binary large object locator variable.*

f) The syntax

```
SQL TYPE IS CLOB AS LOCATOR
```

shall be replaced by

```
Interfaces.SQL.INT
```

in any <Ada CLOB locator variable>. The host variable defined by <Ada CLOB locator variable> is called a *character large object locator variable.*

g) The syntax

```
SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

shall be replaced by

```
Interfaces.SQL.INT
```

in any <Ada user-defined type locator variable>. The host variable defined by <Ada user-defined type locator variable> is called a *user-defined type locator variable*. The data type identified by <path-resolved user-defined type name> is called the *associated user-defined type* of the host variable.

h) The syntax

```
SQL TYPE IS <array type> AS LOCATOR
```

shall be replaced by

```
Interfaces.SQL.INT
```

in any <Ada array locator variable>. The host variable defined by <Ada array locator variable> is called an *array locator variable*. The data type identified by <array type> is called the *associated array type* of the host variable.

i) The syntax

```
SQL TYPE IS <multiset type> AS LOCATOR
```

shall be replaced by

```
Interfaces.SQL.INT
```

in any <Ada multiset locator variable>. The host variable defined by <Ada multiset locator variable> is called a *multiset locator variable*. The data type identified by <multiset type> is called the *associated multiset type* of the host variable.

j) The syntax

```
SQL TYPE IS <reference type>
```

for a given <Ada host identifier> *RTV* shall be replaced by

```
RTV : Interfaces.SQL.CHAR(1..<length>)
```

in any <Ada REF variable>, where <length> is the length in octets of the reference type.

The modified <Ada variable definition> shall be a valid Ada object-declaration in the program derived from the <embedded SQL Ada program>.

6) The reference type identified by <reference type> contained in an <Ada REF variable> is called the *referenced type* of the reference.

7) An <Ada variable definition> shall be specified within the scope of Ada **with** and **use** clauses that specify the following:

```
with Interfaces.SQL;

use Interfaces.SQL;

use Interfaces.SQL.CHARACTER_SET;
```


8) The <character representation> sequence in an <Ada initial value> specifies an initial value to be assigned to the Ada variable. It shall be a valid Ada specification of an initial value.

9) CHAR describes a character string variable whose equivalent SQL data type is CHARACTER with the same length and character set specified by <character set specification>. If <character set specification> is not specified, then an implementation-defined <character set specification> is implicit.

10) SMALLINT, INT, and BIGINT describe exact numeric variables. The equivalent SQL data types are SMALLINT, INTEGER, and BIGINT, respectively.

11) REAL and DOUBLE_PRECISION describe approximate numeric variables. The equivalent SQL data types are REAL and DOUBLE PRECISION, respectively.

12) BOOLEAN describes a boolean variable. The equivalent SQL data type is BOOLEAN.

13) SQLSTATE_TYPE describes a character string variable whose length is the length of the SQLSTATE parameter, five characters.

14) INDICATOR_TYPE describes an exact numeric variable whose specific data type is any <exact numeric type> with a scale of 0 (zero).

Access Rules

None.

General Rules

1) See Subclause 20.1, "<embedded SQL host program>".

Conformance Rules

1) Without Feature B011, "Embedded Ada", conforming SQL language shall not contain an <embedded SQL Ada program>.

2) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain an <Ada BLOB variable>.

3) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain an <Ada CLOB variable>.

4) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain an <Ada BLOB locator variable>.

5) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain an <Ada CLOB locator variable>.

6) Without Feature T071, "BIGINT data type", conforming SQL language shall not contain an <Ada qualified type specification> that contains Interfaces.SQL.BIGINT.

7) Without Feature T071, "BIGINT data type", conforming SQL language shall not contain an <Ada unqualified type specification> that contains BIGINT.

8) Without Feature S241, "Transform functions", conforming SQL language shall not contain an <Ada user-defined type variable>.

9) Without Feature S041, "Basic reference types", conforming SQL language shall not contain an <Ada REF variable>.

10) Without Feature S232, "Array locators", conforming SQL language shall not contain an <Ada array locator variable>.

11) Without Feature S233, "Multiset locators", conforming SQL language shall not contain an <Ada multiset locator variable>.

12) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <path-resolved user-defined type name> simply contained in an <Ada user-defined type locator variable> that identifies a structured type.

20.4 <embedded SQL C program>

Function

Specify an <embedded SQL C program>.

Format

```
<embedded SQL C program> ::= !! See the Syntax Rules.

<C variable definition> ::=
    [ <C storage class> ] [ <C class modifier> ]
    <C variable specification> <semicolon>

<C variable specification> ::=
    <C numeric variable>
  | <C character variable>
  | <C derived variable>

<C storage class> ::=
    auto
  | extern
  | static

<C class modifier> ::=
    const
  | volatile

<C numeric variable> ::=
    { long long | long | short | float | double }
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [ <C initial value> ] }... ]

<C character variable> ::=
    <C character type> [ CHARACTER SET [ IS ] <character set specification> ]
    <C host identifier> <C array specification> [ <C initial value> ]
    [ { <comma> <C host identifier> <C array specification>
    [ <C initial value> ] }... ]

<C character type> ::=
    char
  | unsigned char
  | unsigned short

<C array specification> ::= <left bracket> <length> <right bracket>

<C host identifier> ::= !! See the Syntax Rules.

<C derived variable> ::=
    <C VARCHAR variable>
  | <C NCHAR variable>
  | <C NCHAR VARYING variable>
  | <C CLOB variable>
  | <C NCLOB variable>
```

```
  | <C BLOB variable>
  | <C user-defined type variable>
  | <C CLOB locator variable>
  | <C BLOB locator variable>
  | <C array locator variable>
  | <C multiset locator variable>
  | <C user-defined type locator variable>
  | <C REF variable>

<C VARCHAR variable> ::=
    VARCHAR [ CHARACTER SET [ IS ] <character set specification> ]
    <C host identifier> <C array specification> [ <C initial value> ]
    [ { <comma> <C host identifier> <C array specification> [
    <C initial value> ] }... ]

<C NCHAR variable> ::=
    NCHAR [ CHARACTER SET [ IS ] <character set specification> ]
    <C host identifier> <C array specification> [ <C initial value> ]
    [ { <comma> <C host identifier> <C array specification>
    [ <C initial value> ] } ... ]

<C NCHAR VARYING variable> ::=
    NCHAR VARYING [ CHARACTER SET [ IS ] <character set specification> ]
    <C host identifier> <C array specification> [ <C initial value> ]
    [ { <comma> <C host identifier> <C array specification> [
    <C initial value> ] } ... ]

<C CLOB variable> ::=
    SQL TYPE IS CLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]
    <C host identifier> [ <C initial value> ] [ { <comma> <C host identifier> [
    <C initial value> ] }... ]

<C NCLOB variable> ::=
    SQL TYPE IS NCLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]
    <C host identifier> [ <C initial value> ] [ { <comma> <C host identifier>
    [ <C initial value> ] }... ]

<C user-defined type variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS <predefined type>
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [
    <C initial value> ] } ... ]

<C BLOB variable> ::=
    SQL TYPE IS BLOB <left paren> <large object length> <right paren>
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [
    <C initial value> ] } ... ]

<C CLOB locator variable> ::=
    SQL TYPE IS CLOB AS LOCATOR
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [
    <C initial value> ] } ... ]
```

```
<C BLOB locator variable> ::=
    SQL TYPE IS BLOB AS LOCATOR
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [
    <C initial value> ] } ... ]

<C array locator variable> ::=
    SQL TYPE IS <array type> AS LOCATOR
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [
    <C initial value> ] } ... ]

<C multiset locator variable> ::=
    SQL TYPE IS <multiset type> AS LOCATOR
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [
    <C initial value> ] } ... ]

<C user-defined type locator variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
    <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [
    <C initial value> ] }... ]

<C REF variable> ::=
    SQL TYPE IS <reference type> <C host identifier> [ <C initial value> ]
    [ { <comma> <C host identifier> [ <C initial value> ] }... ]

<C initial value> ::=
    <equals operator> <character representation>...
```

Syntax Rules

1) An <embedded SQL C program> is a compilation unit that consists of C text and SQL text. The C text shall conform to [ISO9899]. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) An <embedded SQL statement> may be specified wherever a C statement may be specified within a function block. If the C statement could include a label prefix, then the <embedded SQL statement> may be immediately preceded by a label prefix.

3) A <C host identifier> is any valid C variable identifier. A <C host identifier> shall be contained in an <embedded SQL C program>.

4) A <C variable definition> defines one or more host variables.

5) A <C variable definition> shall be modified as follows before it is placed into the program derived from the <embedded SQL C program> (see the Syntax Rules of Subclause 20.1, "<embedded SQL host program>"):

   a) Any optional CHARACTER SET specification shall be removed from a <C VARCHAR variable>, a <C character variable>, a <C CLOB variable>, a <C NCHAR variable>, <C NCHAR VARYING variable>, or a <C NCLOB variable>.

   b) The syntax "VARCHAR" shall be replaced by "char" in any <C VARCHAR variable>.

c) The <length> specified in a <C array specification> in any <C character variable> whose <C character type> specifies "`char`" or "`unsigned char`", in any <C VARCHAR variable>, in any <C NCHAR variable>, or in any <C NCHAR VARYING variable>, and the <large object length> specified in a <C CLOB variable> that contains a CHARACTER SET specification or <C NCLOB variable> shall be replaced by a length equal to the length in octets of *PN*, where *PN* is the <C host identifier> specified in the containing <C variable definition>.

NOTE 446 — The <length> does not have to be adjusted for <C character type>s that specify "`unsigned short`" because the units of <length> are already the same units as used by the underlying character set.

d) The syntax "`NCHAR`" in any <C NCHAR variable> and the syntax "`NCHAR VARYING`" in any <C NCHAR VARYING variable> shall be replaced by "`char`".

e) The syntax

```
SQL TYPE IS NCLOB ( L )
```

for a given <C host identifier> *hvn* shall be replaced by

```
struct {
  long          hvn_reserved;
  unsigned long hvn_length;
  char          hvn_data[L];
  } hvn
```

in any <C NCLOB variable>, where *L* is the numeric value of <large object length> as specified in Subclause 5.2, "<token> and <separator>".

f) The syntax

```
SQL TYPE IS CLOB ( L )
```

or the syntax

```
SQL TYPE IS BLOB ( L )
```

for a given <C host identifier> *hvn* shall be replaced by:

```
struct {
  long          hvn_reserved;
  unsigned long hvn_length;
  char          hvn_data[L];
  } hvn
```

in any <C CLOB variable> or <C BLOB variable>, where *L* is the numeric value of <large object length> as specified in Subclause 5.2, "<token> and <separator>".

g) The syntax

```
SQL TYPE IS UDTN AS PDT
```

shall be replaced by

```
ADT
```

in any <C user-defined type variable>, where *ADT* is the data type listed in the "C data type" column corresponding to the row for SQL data type *PDT* in Table 17, "Data type correspondences for C". *ADT*

shall not be "none". The data type identified by *UDTN* is called the *associated user-defined type* of the host variable and the data type identified by *PDT* is called the *associated SQL data type* of the host variable.

h) The syntax

```
SQL TYPE IS BLOB AS LOCATOR
```

shall be replaced by

```
unsigned long
```

in any <C BLOB locator variable>. The host variable defined by <C BLOB locator variable> is called a *binary large object locator variable.*

i) The syntax

```
SQL TYPE IS CLOB AS LOCATOR
```

shall be replaced by

```
unsigned long
```

in any <C CLOB locator variable>. The host variable defined by <C CLOB locator variable> is called a *character large object locator variable.*

j) The syntax

```
SQL TYPE IS <array type> AS LOCATOR
```

shall be replaced by

```
unsigned long
```

in any <C array locator variable>. The host variable defined by <C array locator variable> is called an *array locator variable.* The data type identified by <array type> is called the *associated array type* of the host variable.

k) The syntax

```
SQL TYPE IS <multiset type> AS LOCATOR
```

shall be replaced by

```
unsigned long
```

in any <C multiset locator variable>. The host variable defined by <C multiset locator variable> is called a *multiset locator variable.* The data type identified by <multiset type> is called the *associated multiset type* of the host variable.

l) The syntax

```
SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

shall be replaced by

```
unsigned long
```

in any <C user-defined type locator variable>. The host variable defined by <C user-defined type locator variable> is called a *user-defined type locator variable*. The data type identified by <path-resolved user-defined type name> is called the *associated user-defined type* of the host variable.

m) The syntax

```
SQL TYPE IS <reference type>
```

for a given <C host identifier> *hvn* shall be replaced by

```
unsigned char hvn[L]
```

in any <C REF variable>, where *L* is the length in octets of the reference type.

The modified <C variable definition> shall be a valid C data declaration in the program derived from the <embedded SQL C program>.

6) The reference type identified by <reference type> contained in a <C REF variable> is called the *referenced type* of the reference.

7) The <character representation> sequence contained in a <C initial value> specifies an initial value to be assigned to the C variable. It shall be a valid C specification of an initial value.

8) Except for array specifications for character strings, a <C variable definition> shall specify a scalar type.

9) In a <C variable definition>, the words "VARCHAR", "CHARACTER", "SET", "IS", "VARYING", "BLOB", "CLOB", "NCHAR", "NCLOB", "AS", "LOCATOR", and "REF" may be specified in any combination of upper-case and lower-case letters (see the Syntax Rules of Subclause 5.2, "<token> and <separator>".

10) In a <C character variable>, a <C VARCHAR variable>, or a <C CLOB variable>, if a <character set specification> is specified, then the equivalent SQL data type is CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT whose character set is the same as the character set specified by the <character set specification>. In a <C NCHAR variable>, a <C NCHAR VARYING variable>, or a <C NCLOB variable>, if a <character set specification> is specified, then the equivalent SQL data type is NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, or NATIONAL CHARACTER LARGE OBJECT whose character set is the same as the character set specified by the <character set specification>. If <character set specification> is not specified, then an implementation-defined <character set specification> is implicit.

11) Each <C host identifier> specified in a <C character variable> or a <C NCHAR variable> describes a fixed-length character string. The length is specified by the <length> of the <C array specification>. The value in the host variable is terminated by a null character and the position occupied by this null character is included in the length of the host variable. The equivalent SQL data type is CHARACTER or NATIONAL CHARACTER, respectively, whose length is one less than the <length> of the <C array specification> and whose value does not include the terminating null character. The <length> shall be greater than 1 (one).

12) Each <C host identifier> specified in a <C VARCHAR variable> or a <C NCHAR VARYING variable> describes a variable-length character string. The maximum length is specified by the <length> of the <C array specification>. The value in the host variable is terminated by a null character and the position occupied by this null character is included in the maximum length of the host variable. The equivalent SQL data type is CHARACTER VARYING or NATIONAL CHARACTER VARYING, respectively, whose maximum length is 1 (one) less than the <length> of the <C array specification> and whose value does not include the terminating null character. The <length> shall be greater than 1 (one).

13) "short" describes an exact numeric variable. The equivalent SQL data type is SMALLINT.

14) "long" describes an exact numeric variable. The equivalent SQL data type is INTEGER or BOOLEAN.

15) "long long" describes an exact numeric variable. The equivalent SQL data type is BIGINT.

16) "float" describes an approximate numeric variable. The equivalent SQL data type is REAL.

17) "double" describes an approximate numeric variable. The equivalent SQL data type is DOUBLE PRECISION.

Access Rules

None.

General Rules

1) See Subclause 20.1, "<embedded SQL host program>".

Conformance Rules

1) Without Feature B012, "Embedded C", conforming SQL language shall not contain an <embedded SQL C program>.

2) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <C REF variable>.

3) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <C user-defined type variable>.

4) Without Feature S232, "Array locators", conforming SQL language shall not contain an <C array locator variable>.

5) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <C multiset locator variable>.

6) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <path-resolved user-defined type name> simply contained in a <C user-defined type locator variable> that identifies a structured type.

7) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <C BLOB variable>.

8) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <C CLOB variable>.

9) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <C BLOB locator variable>.

10) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <C CLOB locator variable>.

11) Without Feature T071, "BIGINT data type", conforming SQL language shall not contain a <C numeric variable> that contains `long long`.

20.5 <embedded SQL COBOL program>

Function

Specify an <embedded SQL COBOL program>.

Format

```
<embedded SQL COBOL program> ::= !! See the Syntax Rules.

<COBOL variable definition> ::=
    { 01 |77 } <COBOL host identifier>
    <COBOL type specification> [ <character representation>... ] <period>

<COBOL host identifier> ::= !! See the Syntax Rules.

<COBOL type specification> ::=
    <COBOL character type>
  | <COBOL national character type>
  | <COBOL numeric type>
  | <COBOL integer type>
  | <COBOL derived type specification>

<COBOL derived type specification> ::=
    <COBOL CLOB variable>
  | <COBOL NCLOB variable>
  | <COBOL BLOB variable>
  | <COBOL user-defined type variable>
  | <COBOL CLOB locator variable>
  | <COBOL BLOB locator variable>
  | <COBOL array locator variable>
  | <COBOL multiset locator variable>
  | <COBOL user-defined type locator variable>
  | <COBOL REF variable>

<COBOL character type> ::=
    [ CHARACTER SET [ IS ] <character set specification> ]
    { PIC | PICTURE } [ IS ] { X [ <left paren> <length> <right paren> ] }...

<COBOL national character type> ::=
    [ CHARACTER SET [ IS ] <character set specification> ]
    { PIC | PICTURE } [ IS ] { N [ <left paren> <length> <right paren> ] }...

<COBOL CLOB variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS CLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]

<COBOL NCLOB variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS NCLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]

<COBOL BLOB variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS BLOB <left paren> <large object length> <right paren>
```

```
<COBOL user-defined type variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS <path-resolved user-defined type name>
    AS <predefined type>

<COBOL CLOB locator variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS CLOB AS LOCATOR

<COBOL BLOB locator variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS BLOB AS LOCATOR

<COBOL array locator variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS <array type> AS LOCATOR

<COBOL multiset locator variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS <multiset type> AS LOCATOR

<COBOL user-defined type locator variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR

<COBOL REF variable> ::=
    [ USAGE [ IS ] ] SQL TYPE IS <reference type>

<COBOL numeric type> ::=
    { PIC | PICTURE } [ IS ] S <COBOL nines specification>
    [ USAGE [ IS ] ] DISPLAY SIGN LEADING SEPARATE

<COBOL nines specification> ::=
    <COBOL nines> [ V [ <COBOL nines> ] ]
  | V <COBOL nines>

<COBOL integer type> ::= <COBOL binary integer>

<COBOL binary integer> ::=
    { PIC | PICTURE } [ IS ] S<COBOL nines>
    [ USAGE [ IS ] ] BINARY

<COBOL nines> ::= { 9 [ <left paren> <length> <right paren> ] }...
```

NOTE 447 — The syntax "N(*L*)" is not part of the current COBOL standard, so its use is merely a recommendation; therefore, the production <COBOL national character type> is not normative in this edition of ISO/IEC 9075.

Syntax Rules

1) An <embedded SQL COBOL program> is a compilation unit that consists of COBOL text and SQL text. The COBOL text shall conform to [ISO1989]. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) An <embedded SQL statement> in an <embedded SQL COBOL program> may be specified wherever a COBOL statement may be specified in the Procedure Division of the <embedded SQL COBOL program>. If the COBOL statement could be immediately preceded by a paragraph-name, then the <embedded SQL statement> may be immediately preceded by a paragraph-name.

3) A <COBOL host identifier> is any valid COBOL data-name. A <COBOL host identifier> shall be contained in an <embedded SQL COBOL program>.

4) A <COBOL variable definition> is a restricted form of COBOL data description entry that defines a host variable.

5) A <COBOL variable definition> shall be modified as follows before it is placed into the program derived from the <embedded SQL COBOL program> (see the Syntax Rules of Subclause 20.1, "<embedded SQL host program>").

   a) Any optional CHARACTER SET specification shall be removed from a <COBOL character type>, a <COBOL national character type>, a <COBOL CLOB variable>, and a <COBOL NCLOB variable>.

   b) The <length> specified in any <COBOL character type> and the <large object length> specified in any <COBOL CLOB variable> or <COBOL NCLOB variable> that contains a CHARACTER SET specification shall be replaced by a length equal to the length in octets of *PN*, where *PN* is the <COBOL host identifier> specified in the containing <COBOL variable definition>.

      NOTE 448 — The <length> specified in a <COBOL national character type> does not have to be adjusted, because the units of <length> are already the same units as used by the underlying character set.

      NOTE 449 — The syntax "N(*L*)" is not part of the current COBOL standard, so its use is merely a recommendation; therefore, the production <COBOL national character type> is not normative in ISO/IEC 9075.

   c) The syntax

      ```
      SQL TYPE IS CLOB ( L )
      ```

      or the syntax

      ```
      SQL TYPE IS NCLOB ( L )
      ```

      or the syntax

      ```
      SQL TYPE IS BLOB ( L )
      ```

      for a given <COBOL host identifier> *HVN* shall be replaced by:

      ```
      49 HVN-RESERVED PIC S9(9) USAGE IS BINARY.
      49 HVN-LENGTH PIC S9(9) USAGE IS BINARY.
      49 HVN-DATA PIC X(L).
      ```

      in any <COBOL CLOB variable> or <COBOL BLOB variable>.

   d) The syntax

      ```
      SQL TYPE IS UDTN AS PDT
      ```

      shall be replaced by

      ```
      ADT
      ```

      in any <COBOL user-defined type variable>, where *ADT* is the data type listed in the "COBOL data type" column corresponding to the row for SQL data type *PDT* in Table 18, "Data type correspondences for COBOL". *ADT* shall not be "none". The data type identified by *UDTN* is called the *associated user-defined type* of the host variable and the data type identified by *PDT* is called the *associated SQL data type* of the host variable.

   e) The syntax

      ```
      SQL TYPE IS BLOB AS LOCATOR
      ```

shall be replaced by

```
PIC S9(9) USAGE IS BINARY
```

in any <COBOL BLOB locator variable>. The host variable defined by <COBOL BLOB locator variable> is called a *binary large object locator variable*.

f) The syntax

```
SQL TYPE IS CLOB AS LOCATOR
```

shall be replaced by

```
PIC S9(9) USAGE IS BINARY
```

in any <COBOL CLOB locator variable>. The host variable defined by <COBOL CLOB locator variable> is called a *character large object locator variable*.

g) The syntax

```
SQL TYPE IS <array type> AS LOCATOR
```

shall be replaced by

```
PIC S9(9) USAGE IS BINARY
```

in any <COBOL array locator variable>. The host variable defined by <COBOL array locator variable> is called an *array locator variable*. The data type identified by <array type> is called the *associated array type* of the host variable.

h) The syntax

```
SQL TYPE IS <multiset type> AS LOCATOR
```

shall be replaced by

```
PIC S9(9) USAGE IS BINARY
```

in any <COBOL multiset locator variable>. The host variable defined by <COBOL multiset locator variable> is called a *multiset locator variable*. The data type identified by <multiset type> is called the *associated multiset type* of the host variable.

i) The syntax

```
SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

shall be replaced by

```
PIC S9(9) USAGE IS BINARY
```

in any <COBOL user-defined type locator variable>. The host variable defined by <COBOL user-defined type locator variable> is called a *user-defined type locator variable*. The data type identified by <path-resolved user-defined type name> is called the *associated user-defined type* of the host variable.

j) The syntax

```
SQL TYPE IS <reference type>
```

for a given <COBOL host identifier> *HVN* shall be replaced by

```
01 HVN PICTURE X(L)
```

in any <COBOL REF variable>, where *L* is the length in octets of the reference type.

The modified <COBOL variable definition> shall be a valid data description entry in the Data Division of the program derived from the <embedded SQL COBOL program>.

6) The reference type identified by <reference type> contained in a <COBOL REF variable> is called the *referenced type* of the reference.

7) The optional <character representation> sequence in a <COBOL variable definition> may specify a VALUE clause. Whether other clauses may be specified is implementation-defined. The <character representation> sequence shall be such that the <COBOL variable definition> is a valid COBOL data description entry.

8) A <COBOL character type> describes a character string variable whose equivalent SQL data type is CHARACTER with the same length and character set specified by <character set specification>. If <character set specification> is not specified, then an implementation-defined <character set specification> is implicit.

9) A <COBOL numeric type> describes an exact numeric variable. The equivalent SQL data type is NUMERIC of the same precision and scale.

10) A <COBOL binary integer> describes an exact numeric variable. The equivalent SQL data type is SMALLINT, INTEGER, or BIGINT.

Access Rules

None.

General Rules

1) See Subclause 20.1, "<embedded SQL host program>".

Conformance Rules

1) Without Feature B013, "Embedded COBOL", conforming SQL language shall not contain an <embedded SQL COBOL program>.

2) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <COBOL REF variable>.

3) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <COBOL user-defined type variable>.

4) Without Feature S232, "Array locators", conforming SQL language shall not contain a <COBOL array locator variable>.

5) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <COBOL multiset locator variable>.

6) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <path-resolved user-defined type name> simply contained in a <COBOL user-defined type locator variable> that identifies a structured type.

7) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <COBOL BLOB variable>.

8) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <COBOL CLOB variable>.

9) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <COBOL BLOB locator variable>.

10) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <COBOL CLOB locator variable>.

20.6 <embedded SQL Fortran program>

Function

Specify an <embedded SQL Fortran program>.

Format

```
<embedded SQL Fortran program> ::= !! See the Syntax Rules.

<Fortran variable definition> ::=
    <Fortran type specification> <Fortran host identifier>
    [ { <comma> <Fortran host identifier> }... ]

<Fortran host identifier> ::= !! See the Syntax Rules.

<Fortran type specification> ::=
    CHARACTER [ <asterisk> <length> ] [ CHARACTER SET
    [ IS ] <character set specification> ]
  | CHARACTER KIND = n [ <asterisk> <length> ]
    [ CHARACTER SET [ IS ] <character set specification> ]
  | INTEGER
  | REAL
  | DOUBLE PRECISION
  | LOGICAL
  | <Fortran derived type specification>

<Fortran derived type specification> ::=
    <Fortran CLOB variable>
  | <Fortran BLOB variable>
  | <Fortran user-defined type variable>
  | <Fortran CLOB locator variable>
  | <Fortran BLOB locator variable>
  | <Fortran user-defined type locator variable>
  | <Fortran array locator variable>
  | <Fortran multiset locator variable>
  | <Fortran REF variable>

<Fortran CLOB variable> ::=
    SQL TYPE IS CLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]

<Fortran BLOB variable> ::=
    SQL TYPE IS BLOB <left paren> <large object length> <right paren>

<Fortran user-defined type variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS <predefined type>

<Fortran CLOB locator variable> ::=
    SQL TYPE IS CLOB AS LOCATOR

<Fortran BLOB locator variable> ::=
    SQL TYPE IS BLOB AS LOCATOR
```

```
<Fortran user-defined type locator variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR

<Fortran array locator variable> ::=
    SQL TYPE IS <array type> AS LOCATOR

<Fortran multiset locator variable> ::=
    SQL TYPE IS <multiset type> AS LOCATOR

<Fortran REF variable> ::=
    SQL TYPE IS <reference type>
```

Syntax Rules

1) An <embedded SQL Fortran program> is a compilation unit that consists of Fortran text and SQL text. The Fortran text shall conform to [ISO1539]. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) An <embedded SQL statement> may be specified wherever an executable Fortran statement may be specified. An <embedded SQL statement> that precedes any executable Fortran statement in the containing <embedded SQL Fortran program> shall not have a Fortran statement number. Otherwise, if the Fortran statement could have a statement number then the <embedded SQL statement> can have a statement number.

3) Blanks are significant in <embedded SQL statement>s. The rules for <separator>s in an <embedded SQL statement> are as specified in Subclause 5.2, "<token> and <separator>".

4) A <Fortran host identifier> is any valid Fortran variable name with all <space> characters removed. A <Fortran host identifier> shall be contained in an <embedded SQL Fortran program>.

5) A <Fortran variable definition> is a restricted form of Fortran type-statement that defines one or more host variables.

6) A <Fortran variable definition> shall be modified as follows before it is placed into the program derived from the <embedded SQL Fortran program> (see the Syntax Rules Subclause 20.1, "<embedded SQL host program>").

   a) Any optional CHARACTER SET specification shall be removed from the CHARACTER and the CHARACTER KIND=*n* alternatives in a <Fortran type specification>.

   b) The <length> specified in the CHARACTER alternative of any <Fortran type specification> and the <large object length> specified in any <Fortran CLOB variable> that contains a CHARACTER SET specification shall be replaced by a length equal to the length in octets of *PN*, where *PN* is the <Fortran host identifier> specified in the containing <Fortran variable definition>.

      NOTE 450 — The <length> does not have to be adjusted for CHARACTER KIND=*n* alternatives of any <Fortran type specification>, because the units of <length> are already the same units as used by the underlying character set.

   c) The syntax

      ```
      SQL TYPE IS CLOB ( L )
      ```

      and the syntax

      ```
      SQL TYPE IS BLOB ( L )
      ```


for a given <Fortran host identifier> *HVN* shall be replaced by

```
INTEGER HVN_RESERVED
INTEGER HVN_LENGTH
CHARACTER HVN_DATA [ <asterisk> L ]
```

in any <Fortran CLOB variable> or <Fortran BLOB variable>, where *L* is the numeric value of <large object length> as specified in Subclause 5.2, "<token> and <separator>".

d) The syntax

```
SQL TYPE IS UDTN AS PDT
```

shall be replaced by

```
ADT
```

in any <Fortran user-defined type variable>, where *ADT* is the data type listed in the "Fortran data type" column corresponding to the row for SQL data type *PDT* in Table 19, "Data type correspondences for Fortran". *ADT* shall not be "none". The data type identified by *UDTN* is called the *associated user-defined type* of the host variable and the data type identified by *PDT* is called the *associated SQL data type* of the host variable.

e) The syntax

```
SQL TYPE IS BLOB AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Fortran BLOB locator variable>. The host variable defined by <Fortran BLOB locator variable> is called a *binary large object locator variable.*

f) The syntax

```
SQL TYPE IS CLOB AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Fortran CLOB locator variable>. The host variable defined by <Fortran CLOB locator variable> is called a *character large object locator variable.*

g) The syntax

```
SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Fortran user-defined type locator variable>. The host variable defined by <Fortran user-defined type locator variable> is called a *user-defined type locator variable.* The data type identified by <path-resolved user-defined type name> is called the *associated user-defined type* of the host variable.

h) The syntax

```
SQL TYPE IS <array type> AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Fortran array locator variable>. The host variable defined by <Fortran array locator variable> is called an *array locator variable*. The data type identified by <array type> is called the *associated array type* of the host variable.

i) The syntax

```
SQL TYPE IS <multiset type> AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Fortran multiset locator variable>. The host variable defined by <Fortran multiset locator variable> is called a *multiset locator variable*. The data type identified by <multiset type> is called the *associated multiset type* of the host variable.

j) The syntax

```
SQL TYPE IS <reference type>
```

for a given <Fortran host identifier> *HVN* shall be replaced by

```
CHARACTER HVN * <length>
```

in any <Fortran REF variable>, where <length> is the length in octets of the reference type.

The modified <Fortran variable definition> shall be a valid Fortran type-statement in the program derived from the <embedded SQL Fortran program>.

7) The reference type identified by <reference type> contained in an <Fortran REF variable> is called the *referenced type* of the reference.

8) CHARACTER without "KIND=*n*" describes a character string variable whose equivalent SQL data type is CHARACTER with the same length and character set specified by <character set specification>. If <character set specification> is not specified, then an implementation-defined <character set specification> is implicit.

9) CHARACTER KIND=*n* describes a character string variable whose equivalent SQL data type is either CHARACTER or NATIONAL CHARACTER with the same length and character set specified by <character set specification>. If <character set specification> is not specified, then an implementation-defined <character set specification> is implicit. The value of *n* determines implementation-defined characteristics of the Fortran variable; values of *n* are implementation-defined.

10) INTEGER describes an exact numeric variable. The equivalent SQL data type is INTEGER.

11) REAL describes an approximate numeric variable. The equivalent SQL data type is REAL.

12) DOUBLE PRECISION describes an approximate numeric variable. The equivalent SQL data type is DOUBLE PRECISION.

13) LOGICAL describes a boolean variable. The equivalent SQL data type is BOOLEAN.

Access Rules

None.

General Rules

1) See Subclause 20.1, "<embedded SQL host program>".

Conformance Rules

1) Without Feature B014, "Embedded Fortran", conforming SQL language shall not contain an <embedded SQL Fortran program>.

2) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <Fortran REF variable>.

3) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <Fortran user-defined type variable>.

4) Without Feature S232, "Array locators", conforming SQL language shall not contain a <Fortran array locator variable>.

5) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <Fortran multiset locator variable>.

6) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <path-resolved user-defined type name> simply contained in a <Fortran user-defined type locator variable> that identifies a structured type.

7) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Fortran BLOB variable>.

8) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Fortran CLOB variable>.

9) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Fortran BLOB locator variable>.

10) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Fortran CLOB locator variable>.

20.7 <embedded SQL MUMPS program>

Function

Specify an <embedded SQL MUMPS program>.

Format

```
<embedded SQL MUMPS program> ::= !! See the Syntax Rules.

<MUMPS variable definition> ::=
    <MUMPS numeric variable> <semicolon>
  | <MUMPS character variable> <semicolon>
  | <MUMPS derived type specification> <semicolon>

<MUMPS character variable> ::=
    VARCHAR <MUMPS host identifier> <MUMPS length specification>
    [ { <comma> <MUMPS host identifier> <MUMPS length specification> }... ]

<MUMPS host identifier> ::= !! See the Syntax Rules.

<MUMPS length specification> ::= <left paren> <length> <right paren>

<MUMPS numeric variable> ::=
    <MUMPS type specification> <MUMPS host identifier>
    [ { <comma> <MUMPS host identifier> }... ]

<MUMPS type specification> ::=
    INT
  | DEC [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | REAL

<MUMPS derived type specification> ::=
    <MUMPS CLOB variable>
  | <MUMPS BLOB variable>
  | <MUMPS user-defined type variable>
  | <MUMPS CLOB locator variable>
  | <MUMPS BLOB locator variable>
  | <MUMPS user-defined type locator variable>
  | <MUMPS array locator variable>
  | <MUMPS multiset locator variable>
  | <MUMPS REF variable>

<MUMPS CLOB variable> ::=
    SQL TYPE IS CLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]

<MUMPS BLOB variable> ::=
    SQL TYPE IS BLOB <left paren> <large object length> <right paren>

<MUMPS user-defined type variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS <predefined type>

<MUMPS CLOB locator variable> ::=
```

```
    SQL TYPE IS CLOB AS LOCATOR

<MUMPS BLOB locator variable> ::=
    SQL TYPE IS BLOB AS LOCATOR

<MUMPS user-defined type locator variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR

<MUMPS array locator variable> ::=
    SQL TYPE IS <array type> AS LOCATOR

<MUMPS multiset locator variable> ::=
    SQL TYPE IS <multiset type> AS LOCATOR

<MUMPS REF variable> ::=
    SQL TYPE IS <reference type>
```

Syntax Rules

1) An <embedded SQL MUMPS program> is a compilation unit that consists of M text and SQL text. The M text shall conform to [ISO11756]. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) A <MUMPS host identifier> is any valid M variable name. A <MUMPS host identifier> shall be contained in an <embedded SQL MUMPS program>.

3) An <embedded SQL statement> may be specified wherever an M command may be specified.

4) A <MUMPS variable definition> defines one or more host variables.

5) The <MUMPS character variable> describes a variable-length character string. The equivalent SQL data type is CHARACTER VARYING whose maximum length is the <length> of the <MUMPS length specification> and whose character set is implementation-defined.

6) INT describes an exact numeric variable. The equivalent SQL data type is INTEGER.

7) DEC describes an exact numeric variable. The <scale> shall not be greater than the <precision>. The equivalent SQL data type is DECIMAL with the same <precision> and <scale>.

8) REAL describes an approximate numeric variable. The equivalent SQL data type is REAL.

9) A <MUMPS derived type specification> shall be modified as follows before it is placed into the program derived from the <embedded SQL MUMPS program> (see the Syntax Rules of Subclause 20.1, "<embedded SQL host program>").

   a) Any optional CHARACTER SET specification shall be removed from a <MUMPS CLOB variable>.

   b) The syntax

      ```
      SQL TYPE IS CLOB ( L )
      ```

      and the syntax

      ```
      SQL TYPE IS BLOB ( L )
      ```

      for a given <MUMPS host identifier> *HVN* shall be replaced by

```
INT HVN_RESERVED
INT HVN_LENGTH
VARCHAR HVN_DATA L
```

in any <MUMPS CLOB variable> or <MUMPS BLOB variable>, where *L* is the numeric value of <large object length> as specified in Subclause 5.2, "<token> and <separator>".

c) The syntax

```
SQL TYPE IS UDTN AS PDT
```

shall be replaced by

```
ADT
```

in any <MUMPS user-defined type variable>, where *ADT* is the data type listed in the "MUMPS data type" column corresponding to the row for SQL data type *PDT* in Table 20, "Data type correspondences for M", *ADT* shall not be "none". The data type identified by *UDTN* is called the *associated user-defined type* of the host variable and the data type identified by *PDT* is called the *associated SQL data type* of the host variable.

d) The syntax

```
SQL TYPE IS BLOB AS LOCATOR
```

shall be replaced by

```
INT
```

in any or <MUMPS BLOB locator variable>. The host variable defined by <MUMPS BLOB locator variable> is called a *binary large object locator variable.*

e) The syntax

```
SQL TYPE IS CLOB AS LOCATOR
```

shall be replaced by

```
INT
```

in any <MUMPS CLOB locator variable>. The host variable defined by <MUMPS CLOB locator variable> is called a *character large object locator variable.*

f) The syntax

```
SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

shall be replaced by

```
INT
```

in any <MUMPS user-defined type locator variable>. The host variable defined by <MUMPS user-defined type locator variable> is called a *user-defined type locator variable.* The data type identified by <path-resolved user-defined type name> is called the *associated user-defined type* of the host variable.

g) The syntax

```
SQL TYPE IS <array type> AS LOCATOR
```

shall be replaced by

```
INT
```

in any <MUMPS array locator variable>. The host variable defined by <MUMPS array locator variable> is called an *array locator variable*. The data type identified by <array type> is called the *associated array type* of the host variable.

h) The syntax

```
SQL TYPE IS <multiset type> AS LOCATOR
```

shall be replaced by

```
INT
```

in any <MUMPS multiset locator variable>. The host variable defined by <MUMPS multiset locator variable> is called a *multiset locator variable*. The data type identified by <multiset type> is called the *associated multiset type* of the host variable.

i) The syntax

```
SQL TYPE IS <reference type>
```

for a given <MUMPS host identifier> *HVN* shall be replaced by

```
VARCHAR HVN L
```

in any <MUMPS REF variable>, where *L* is the length in octets of the reference type.

The modified <MUMPS variable definition> shall be a valid M variable in the program derived from the <embedded SQL MUMPS program>.

10) The reference type identified by <reference type> contained in an <MUMPS REF variable> is called the *referenced type* of the reference.

Access Rules

None.

General Rules

1) See Subclause 20.1, "<embedded SQL host program>".

Conformance Rules

1) Without Feature B015, "Embedded MUMPS", conforming SQL language shall not contain an <embedded SQL MUMPS program>.

2) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <MUMPS REF variable>.

3) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <MUMPS user-defined type variable>.

4) Without Feature S232, "Array locators", conforming SQL language shall not contain a <MUMPS array locator variable>.

5) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <MUMPS multiset locator variable>.

6) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <path-resolved user-defined type name> simply contained in a <MUMPS user-defined type locator variable> that identifies a structured type.

7) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <MUMPS BLOB variable>.

8) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <MUMPS CLOB variable>.

9) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <MUMPS BLOB locator variable>.

10) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a and <MUMPS CLOB locator variable>.

20.8 <embedded SQL Pascal program>

Function

Specify an <embedded SQL Pascal program>.

Format

```
<embedded SQL Pascal program> ::= !! See the Syntax Rules.

<Pascal variable definition> ::=
    <Pascal host identifier> [ { <comma> <Pascal host identifier> }... ] <colon>
    <Pascal type specification> <semicolon>

<Pascal host identifier> ::= !! See the Syntax Rules.

<Pascal type specification> ::=
    PACKED ARRAY <left bracket> 1 <double period> <length> <right bracket>
    OF CHAR [ CHARACTER SET [ IS ] <character set specification> ]
  | INTEGER
  | REAL
  | CHAR [ CHARACTER SET [ IS ] <character set specification> ]
  | BOOLEAN
  | <Pascal derived type specification>

<Pascal derived type specification> ::=
    <Pascal CLOB variable>
  | <Pascal BLOB variable>
  | <Pascal user-defined type variable>
  | <Pascal CLOB locator variable>
  | <Pascal BLOB locator variable>
  | <Pascal user-defined type locator variable>
  | <Pascal array locator variable>
  | <Pascal multiset locator variable>
  | <Pascal REF variable>

<Pascal CLOB variable> ::=
    SQL TYPE IS CLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]

<Pascal BLOB variable> ::=
    SQL TYPE IS BLOB <left paren> <large object length> <right paren>

<Pascal CLOB locator variable> ::=
    SQL TYPE IS CLOB AS LOCATOR

<Pascal user-defined type variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS <predefined type>

<Pascal BLOB locator variable> ::=
    SQL TYPE IS BLOB AS LOCATOR

<Pascal user-defined type locator variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

```
<Pascal array locator variable> ::=
    SQL TYPE IS <array type> AS LOCATOR

<Pascal multiset locator variable> ::=
    SQL TYPE IS <multiset type> AS LOCATOR

<Pascal REF variable> ::=
    SQL TYPE IS <reference type>
```

Syntax Rules

1) An <embedded SQL Pascal program> is a compilation unit that consists of Pascal text and SQL text. The Pascal text shall conform to one of [ISO7185] or [ISO10206]. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) An <embedded SQL statement> may be specified wherever a Pascal statement may be specified. An <embedded SQL statement> may be prefixed by a Pascal label.

3) A <Pascal host identifier> is a Pascal variable-identifier whose applied instance denotes a defining instance within an <embedded SQL begin declare> and an <embedded SQL end declare>.

4) A <Pascal variable definition> defines one or more <Pascal host identifier>s.

5) A <Pascal variable definition> shall be modified as follows before it is placed into the program derived from the <embedded SQL Pascal program> (see the Syntax Rules of Subclause 20.1, "<embedded SQL host program>").

   a) Any optional CHARACTER SET specification shall be removed from the PACKED ARRAY OF CHAR or CHAR alternatives of a <Pascal type specification> and a <Pascal CLOB variable>.

   b) The <length> specified in the PACKED ARRAY OF CHAR alternative of any <Pascal type specification> that contains a CHARACTER SET specification and the <large object length> specified in a <Pascal CLOB variable> that contains a CHARACTER SET specification shall be replaced by a length equal to the length in octets of *PN*, where *PN* is the <Pascal host identifier> specified in the containing <Pascal variable definition>.

   c) If any <Pascal type specification> specifies the syntax "CHAR" and contains a CHARACTER SET specification, then let *L* be a length equal to the length in octets of *PN* and *PN* be the <Pascal host identifier> specified in the containing <Pascal variable definition>. If *L* is greater than 1 (one), then "CHAR" shall be replaced by "PACKED ARRAY [1..*L*] OF CHAR".

   d) The syntax

      ```
      SQL TYPE IS CLOB ( L )
      ```

      and the syntax

      ```
      SQL TYPE IS BLOB ( L )
      ```

      for a given <Pascal host identifier> *HVN* shall be replaced by

      ```
      VAR HVN = RECORD
        HVN_RESERVED : INTEGER;
        HVN_LENGTH : INTEGER;
      ```

```
    HVN_DATA : PACKED ARRAY [ 1..L ] OF CHAR;
END;
```

in any <Pascal CLOB variable> or <Pascal BLOB variable>, where *L* is the numeric value of <large object length> as specified in Subclause 5.2, "<token> and <separator>".

e) The syntax

```
SQL TYPE IS UDTN AS PDT
```

shall be replaced by

```
ADT
```

in any <Pascal user-defined type variable>, where *ADT* is the data type listed in the "Pascal data type" column corresponding to the row for SQL data type *PDT* in Table 21, "Data type correspondences for Pascal". *ADT* shall not be "none". The data type identified by *UDTN* is called the *associated user-defined type* of the host variable and the data type identified by *PDT* is called the *associated SQL data type* of the host variable.

f) The syntax

```
SQL TYPE IS BLOB AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Pascal BLOB locator variable>. The host variable defined by <Pascal BLOB locator variable> is called a *binary large object locator variable.*

g) The syntax

```
SQL TYPE IS CLOB AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Pascal CLOB locator variable>. The host variable defined by <Pascal CLOB locator variable> is called a *character large object locator variable.*

h) The syntax

```
SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Pascal user-defined type locator variable>. The host variable defined by <Pascal user-defined type locator variable> is called a *user-defined type locator variable.* The data type identified by <path-resolved user-defined type name> is called the *associated user-defined type* of the host variable.

i) The syntax

```
SQL TYPE IS <array type> AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Pascal array locator variable>. The host variable defined by <Pascal array locator variable> is called an *array locator variable*. The data type identified by <array type> is called the *associated array type* of the host variable.

j) The syntax

```
SQL TYPE IS <multiset type> AS LOCATOR
```

shall be replaced by

```
INTEGER
```

in any <Pascal multiset locator variable>. The host variable defined by <Pascal multiset locator variable> is called a *multiset locator variable*. The data type identified by <multiset type> is called the *associated multiset type* of the host variable.

k) The syntax

```
SQL TYPE IS <reference type>
```

for a given <Pascal host identifier> *HVN* shall be replaced by

```
HVN : PACKED ARRAY [1..<length>] OF CHAR
```

in any <Pascal REF variable>, where <length> is the length in octets of the reference type.

The modified <Pascal variable definition> shall be a valid Pascal variable-declaration in the program derived from the <embedded SQL Pascal program>.

6) The reference type identified by <reference type> contained in an <Pascal REF variable> is called the *referenced type* of the reference.

7) CHAR specified without a CHARACTER SET specification is the ordinal-type-identifier of PASCAL. The equivalent SQL data type is CHARACTER with length 1 (one).

8) PACKED ARRAY [1..<length>] OF CHAR describes a character string having 2 or more components of the simple type CHAR. The equivalent SQL data type is CHARACTER with the same length and character set specified by <character set specification>. If <character set specification> is not specified, then an implementation-defined <character set specification> is implicit.

9) INTEGER describes an exact numeric variable. The equivalent SQL data type is INTEGER.

10) REAL describes an approximate numeric variable. The equivalent SQL data type is REAL.

11) BOOLEAN describes a boolean variable. The equivalent SQL data type is BOOLEAN.

Access Rules

None.

General Rules

1) See Subclause 20.1, "<embedded SQL host program>".

Conformance Rules

1) Without Feature B016, "Embedded Pascal", conforming SQL language shall not contain an <embedded SQL Pascal program>.

2) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <Pascal REF variable>.

3) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <Pascal user-defined type variable>.

4) Without Feature S232, "Array locators", conforming SQL language shall not contain a <Pascal array locator variable>.

5) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <Pascal multiset locator variable>.

6) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <path-resolved user-defined type name> simply contained in a <Pascal user-defined type locator variable> that identifies a structured type.

7) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Pascal BLOB variable>.

8) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Pascal CLOB variable>.

9) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Pascal BLOB locator variable>.

10) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <Pascal BLOB variable>, <Pascal CLOB variable>, <Pascal CLOB locator variable>.

20.9 <embedded SQL PL/I program>

Function

Specify an <embedded SQL PL/I program>.

Format

```
<embedded SQL PL/I program> ::= !! See the Syntax Rules.

<PL/I variable definition> ::=
    { DCL | DECLARE } <PL/I type specification> [ <character representation>... ] <semicolon>
  | { <PL/I host identifier> | <left paren> <PL/I host identifier>
    [ { <comma> <PL/I host identifier> }... ] <right paren> }
    <PL/I type specification> [ <character representation>... ] <semicolon>

<PL/I host identifier> ::= !! See the Syntax Rules.

<PL/I type specification> ::=
    { CHAR | CHARACTER } [ VARYING ] <left paren> <length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]
  | <PL/I type fixed decimal> <left paren> <precision> [ <comma> <scale> ] <right paren>
  | <PL/I type fixed binary> [ <left paren> <precision> <right paren> ]
  | <PL/I type float binary> <left paren> <precision> <right paren>
  | <PL/I derived type specification>

<PL/I derived type specification> ::=
    <PL/I CLOB variable>
  | <PL/I BLOB variable>
  | <PL/I user-defined type variable>
  | <PL/I CLOB locator variable>
  | <PL/I BLOB locator variable>
  | <PL/I user-defined type locator variable>
  | <PL/I array locator variable>
  | <PL/I multiset locator variable>
  | <PL/I REF variable>

<PL/I CLOB variable> ::=
    SQL TYPE IS CLOB <left paren> <large object length> <right paren>
    [ CHARACTER SET [ IS ] <character set specification> ]

<PL/I BLOB variable> ::=
    SQL TYPE IS BLOB <left paren> <large object length> <right paren>

<PL/I user-defined type variable> ::=
    SQL TYPE IS <path-resolved user-defined type name> AS <predefined type>

<PL/I CLOB locator variable> ::=
    SQL TYPE IS CLOB AS LOCATOR

<PL/I BLOB locator variable> ::=
    SQL TYPE IS BLOB AS LOCATOR

<PL/I user-defined type locator variable> ::=
```

```
    SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR

<PL/I array locator variable> ::=
    SQL TYPE IS <array type> AS LOCATOR

<PL/I multiset locator variable> ::=
    SQL TYPE IS <multiset type> AS LOCATOR

<PL/I REF variable> ::=
    SQL TYPE IS <reference type>

<PL/I type fixed decimal> ::=
    { DEC | DECIMAL } FIXED
  | FIXED { DEC | DECIMAL }

<PL/I type fixed binary> ::=
    { BIN | BINARY } FIXED
  | FIXED { BIN | BINARY }

<PL/I type float binary> ::=
    { BIN | BINARY } FLOAT
  | FLOAT { BIN | BINARY }
```

Syntax Rules

1) An <embedded SQL PL/I program> is a compilation unit that consists of PL/I text and SQL text. The PL/I text shall conform to [ISO6160]. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) An <embedded SQL statement> may be specified wherever a PL/I statement may be specified within a procedure block. If the PL/I statement could include a label prefix, the <embedded SQL statement> may be immediately preceded by a label prefix.

3) A <PL/I host identifier> is any valid PL/I variable identifier. A <PL/I host identifier> shall be contained in an <embedded SQL PL/I program>.

4) A <PL/I variable definition> defines one or more host variables.

5) A <PL/I variable definition> shall be modified as follows before it is placed into the program derived from the <embedded SQL PL/I program> (see the Syntax Rules of Subclause 20.1, "<embedded SQL host program>").

   a) Any optional CHARACTER SET specification shall be removed from the CHARACTER or CHARACTER VARYING alternatives of a <PL/I type specification>.

   b) The <length> specified in the CHARACTER or CHARACTER VARYING alternatives of any <PL/I type specification> or a <PL/I CLOB variable> that contains a CHARACTER SET specification shall be replaced by a length equal to the length in octets of *PN*, where *PN* is the <PL/I host identifier> specified in the containing <PL/I variable definition>.

   c) The syntax

      ```
      SQL TYPE IS CLOB ( L )
      ```

      and the syntax

```
SQL TYPE IS BLOB ( L )
```

for a given <PL/I host identifier> *HVN* shall be replaced by

```
DCL 1 HVN
      2 HVN_RESERVED FIXED BINARY(31),
      2 HVN_LENGTH   FIXED BINARY(31),
      2 HVN_DATA     CHARACTER(<length>);
```

in any <PL/I CLOB variable> or <PL/I BLOB variable>, where *L* is the numeric value of <large object length> as specified in Subclause 5.2, "<token> and <separator>".

d) The syntax

```
SQL TYPE IS UDTN AS PDT
```

shall be replaced by

ADT

in any <PL/I user-defined type variable>, where *ADT* is the data type listed in the "PL/I data type" column corresponding to the row for SQL data type *PDT* in Table 22, "Data type correspondences for PL/I". *ADT* shall not be "none". The data type identified by *UDTN* is called the *associated user-defined type* of the host variable and the data type identified by *PDT* is called the *associated SQL data type* of the host variable.

e) The syntax

```
SQL TYPE IS BLOB AS LOCATOR
```

shall be replaced by

```
FIXED BINARY(31)
```

in any <PL/I BLOB locator variable>. The host variable defined by <PL/I BLOB locator variable> is called a *binary large object locator variable.*

f) The syntax

```
SQL TYPE IS CLOB AS LOCATOR
```

shall be replaced by

```
FIXED BINARY(31)
```

in any <PL/I CLOB locator variable>. The host variable defined by <PL/I CLOB locator variable> is called a *character large object locator variable.*

g) The syntax

```
SQL TYPE IS <path-resolved user-defined type name> AS LOCATOR
```

shall be replaced by

```
FIXED BINARY(31)
```

in any <PL/I user-defined type locator variable>. The host variable defined by <PL/I user-defined type locator variable> is called a *user-defined type locator variable*. The data type identified by <path-resolved user-defined type name> is called the *associated user-defined type* of the host variable.

h) The syntax

```
SQL TYPE IS <array type> AS LOCATOR
```

shall be replaced by

```
FIXED BINARY(31)
```

in any <PL/I array locator variable>. The host variable defined by <PL/I array locator variable> is called an *array locator variable*. The data type identified by <array type> is called the *associated array type* of the host variable.

i) The syntax

```
SQL TYPE IS <multiset type> AS LOCATOR
```

shall be replaced by

```
FIXED BINARY(31)
```

in any <PL/I multiset locator variable>. The host variable defined by <PL/I multiset locator variable> is called a *multiset locator variable*. The data type identified by <multiset type> is called the *associated multiset type* of the host variable.

j) The syntax

```
SQL TYPE IS <reference type>
```

for a given <PL/I host identifier> *HVN* shall be replaced by

```
DCL HVN CHARACTER(<length>) VARYING
```

in any <PL/I REF variable>, where <length> is the length in octets of the reference type.

The modified <PL/I variable definition> shall be a valid PL/I data declaration in the program derived from the <embedded SQL PL/I program>.

6) The reference type identified by <reference type> contained in an <PL/I REF variable> is called the *referenced type* of the reference.

7) A <PL/I variable definition> shall specify a scalar variable, not an array or structure.

8) The optional <character representation> sequence in a <PL/I variable definition> may specify an INITIAL clause. Whether other clauses may be specified is implementation-defined. The <character representation> sequence shall be such that the <PL/I variable definition> is a valid PL/I DECLARE statement.

9) CHARACTER describes a character string variable whose equivalent SQL data type has the character set specified by <character set specification>. If <character set specification> is not specified, then an implementation-defined <character set specification> is implicit.

Case:

a) If VARYING is not specified, then the length of the variable is fixed. The equivalent SQL data type is CHARACTER with the same length.

b) If VARYING is specified, then the variable is of variable length, with maximum size the value of <length>. The equivalent SQL data type is CHARACTER VARYING with the same maximum length.

10) FIXED DECIMAL describes an exact numeric variable. The <scale> shall not be greater than the <precision>. The equivalent SQL data type is DECIMAL with the same <precision> and <scale>.

11) FIXED BINARY describes an exact numeric variable. The equivalent SQL data type is SMALLINT, INTEGER, or BIGINT.

12) FLOAT BINARY describes an approximate numeric variable. The equivalent SQL data type is FLOAT with the same <precision>.

Access Rules

None.

General Rules

1) See Subclause 20.1, "<embedded SQL host program>"

Conformance Rules

1) Without Feature B017, "Embedded PL/I", conforming SQL language shall not contain an <embedded SQL PL/I program>.

2) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <PL/I REF variable>.

3) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <PL/I user-defined type variable>.

4) Without Feature S232, "Array locators", conforming SQL language shall not contain a <PL/I array locator variable>.

5) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <PL/I multiset locator variable>.

6) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <path-resolved user-defined type name> simply contained in a <PL/I user-defined type locator variable> that identifies a structured type.

7) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <PL/I BLOB variable>.

8) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <PL/I CLOB variable>.

9) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <PL/I BLOB locator variable>.

10) Without Feature T041, "Basic LOB data type support", conforming SQL language shall not contain a <PL/I CLOB locator variable>.

This page intentionally left blank.

21 Direct invocation of SQL

21.1 <direct SQL statement>

Function

Specify direct execution of SQL.

Format

```
<direct SQL statement> ::= <directly executable statement> <semicolon>

<directly executable statement> ::=
    <direct SQL data statement>
  | <SQL schema statement>
  | <SQL transaction statement>
  | <SQL connection statement>
  | <SQL session statement>
  | <direct implementation-defined statement>

<direct SQL data statement> ::=
    <delete statement: searched>
  | <direct select statement: multiple rows>
  | <insert statement>
  | <update statement: searched>
  | <merge statement>
  | <temporary table declaration>

<direct implementation-defined statement> ::= !! See the Syntax Rules
```

Syntax Rules

1) The <direct SQL data statement> shall not contain an SQL parameter reference, SQL variable reference, <dynamic parameter specification>, or <embedded variable specification>.

2) The <value specification> that represents the null value is implementation-defined.

3) The Format and Syntax Rules for <direct implementation-defined statement> are implementation-defined.

Access Rules

1) The Access Rules for <direct implementation-defined statement> are implementation-defined.

General Rules

1) The following <direct SQL statement>s are transaction-initiating <direct SQL statement>s:

   a) <direct SQL statement>s that are transaction-initiating <SQL procedure statement>s.

   b) <direct select statement: multiple rows>.

   c) <direct implementation-defined statement>s that are transaction-initiating.

2) After the last invocation of an SQL-statement by an SQL-agent in an SQL-session:

   a) A <rollback statement> or a <commit statement> is effectively executed. If an unrecoverable error has occurred, or if the direct invocation of SQL terminated unexpectedly, or if any constraint is not satisfied, then a <rollback statement> is performed. Otherwise, the choice of which of these SQL-statements to perform is implementation-dependent. The determination of whether a direct invocation of SQL has terminated unexpectedly is implementation-dependent.

   b) Let *D* be the <descriptor name> of any SQL descriptor area that is currently allocated within the current SQL-session. A <deallocate descriptor statement> that specifies

      ```
      DEALLOCATE DESCRIPTOR D
      ```

      is effectively executed.

   c) All SQL-sessions associated with the SQL-agent are terminated.

3) Let *S* be the <direct SQL statement>.

4) A copy of the top cell of the authorization stack is pushed onto the authorization stack.

5) If *S* does not conform to the Format, Syntax Rules, and Access Rules for a <direct SQL statement>, then an exception condition is raised: *syntax error or access rule violation*.

6) When *S* is invoked by the SQL-agent:

   Case:

   a) If *S* is an <SQL connection statement>, then:

      i) The first diagnostics area is emptied.

      ii) *S* is executed.

      iii) If *S* successfully initiated or resumed an SQL-session, then subsequent invocations of a <direct SQL statement> by the SQL-agent are associated with that SQL-session until the SQL-agent terminates the SQL-session or makes it dormant.

   b) Otherwise:

      i) If no SQL-session is current for the SQL-agent, then

         Case:

         1) If the SQL-agent has not executed an <SQL connection statement> and there is no default SQL-session associated with the SQL-agent, then the following <connect statement> is effectively executed:

```
CONNECT TO DEFAULT
```

2) If the SQL-agent has not executed an <SQL connection statement> and there is a default SQL-session associated with the SQL-agent, then the following <set connection statement> is effectively executed:

```
SET CONNECTION DEFAULT
```

3) Otherwise, an exception condition is raised: *connection exception — connection does not exist.*

Subsequent calls to an <externally-invoked procedure> or invocations of a <direct SQL statement> by the SQL-agent are associated with the SQL-session until the SQL-agent terminates the SQL-session or makes it dormant.

ii) If an SQL-transaction is active for the SQL-agent, then *S* is associated with that SQL-transaction. If *S* is a <direct implementation-defined statement>, then it is implementation-defined whether or not *S* may be associated with an active SQL-transaction; if not, then an exception condition is raised: *invalid transaction state — active SQL-transaction.*

iii) If no SQL-transaction is active for the SQL-agent, then

1) Case:

A) If *S* is a transaction-initiating <direct SQL statement>, then an SQL-transaction is initiated.

B) If *S* is a <direct implementation-defined statement>, then it is implementation-defined whether or not *S* initiates an SQL-transaction. If an implementation defines *S* to be transaction-initiating, then an SQL-transaction is initiated.

2) If *S* initiated an SQL-transaction, then:

A) Let *T* be the SQL-transaction initiated by *S*.

B) *T* is associated with this invocation and any subsequent invocations of <direct SQL statement>s or calls to an <externally-invoked procedure> by the SQL-agent until the SQL-agent terminates *T*.

C) If *S* is not a <start transaction statement>, then

Case:

I) If a <set transaction statement> has been executed since the termination of the last SQL-transaction in the SQL-session (or if there has been no previous SQL-transaction in the SQL-session and a <set transaction statement> has been executed), then the access mode, constraint mode, and isolation level of *T* are set as specified by the <set transaction statement>.

II) Otherwise, the access mode, constraint mode for all constraints, and isolation level for *T* are *read-write*, *immediate*, and SERIALIZABLE, respectively.

D) *T* is associated with the SQL-session.

iv) If *S* contains an <SQL schema statement> and the access mode of the current SQL-transaction is *read-only*, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction.*

v) The first diagnostics area is emptied.

vi) *S* is executed.

7) Upon completion of execution, the top cell in the authorization stack is removed.

8) If the execution of a <direct SQL data statement> occurs within the same SQL-transaction as the execution of an SQL-schema statement and this is not allowed by the SQL-implementation, then an exception condition is raised: *invalid transaction state — schema and data statement mixing not supported.*

9) Case:

a) If *S* executed successfully, then either a completion condition is raised: *successful completion*, or a completion condition is raised: *warning*, or a completion condition is raised: *no data.*

b) If *S* did not execute successfully, then all changes made to SQL-data or schemas by the execution of *S* are canceled and an exception condition is raised.

NOTE 451 — The method of raising a condition is implementation-defined.

10) Diagnostics information resulting from the execution of *S* is placed into the first diagnostics area, causing the first condition area in the first diagnostics area to become occupied.

NOTE 452 — The method of accessing the diagnostics information is implementation-defined, but does not alter the contents of the diagnostics area.

Conformance Rules

1) Without Feature B021, "Direct SQL", conforming SQL language shall not contain a <direct SQL statement>.

21.2 <direct select statement: multiple rows>

Function

Specify a statement to retrieve multiple rows from a specified table.

Format

```
<direct select statement: multiple rows> ::= <cursor specification>
```

Syntax Rules

1) The <query expression> or <order by clause> of a <direct select statement: multiple rows> shall not contain a <value specification> other than a <literal>, CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_PATH, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANSFORM_GROUP_FOR_TYPE.

2) The <cursor specification> shall not contain an <updatability clause>.

Access Rules

None.

General Rules

1) Let *Q* be the result of the <cursor specification>.

2) Case:

   a) If *Q* is empty, then a completion condition is raised: *no data*.

   b) Otherwise, *Q* is not empty and *Q* is returned. The method of returning *Q* is implementation-defined.

Conformance Rules

None.

This page intentionally left blank.

22 Diagnostics management

22.1 <get diagnostics statement>

Function

Get exception or completion condition information from a diagnostics area.

Format

```
<get diagnostics statement> ::=
    GET DIAGNOSTICS <SQL diagnostics information>

<SQL diagnostics information> ::=
    <statement information>
  | <condition information>

<statement information> ::=
    <statement information item> [ { <comma> <statement information item> }... ]

<statement information item> ::=
    <simple target specification> <equals operator> <statement information item name>

<statement information item name> ::=
    NUMBER
  | MORE
  | COMMAND_FUNCTION
  | COMMAND_FUNCTION_CODE
  | DYNAMIC_FUNCTION
  | DYNAMIC_FUNCTION_CODE
  | ROW_COUNT
  | TRANSACTIONS_COMMITTED
  | TRANSACTIONS_ROLLED_BACK
  | TRANSACTION_ACTIVE

<condition information> ::=
    { EXCEPTION | CONDITION } <condition number> <condition information item>
    [ { <comma> <condition information item> }... ]

<condition information item> ::=
    <simple target specification> <equals operator> <condition information item name>

<condition information item name> ::=
    CATALOG_NAME
  | CLASS_ORIGIN
  | COLUMN_NAME
  | CONDITION_NUMBER
  | CONNECTION_NAME
  | CONSTRAINT_CATALOG
```

```
| CONSTRAINT_NAME
| CONSTRAINT_SCHEMA
| CURSOR_NAME
| MESSAGE_LENGTH
| MESSAGE_OCTET_LENGTH
| MESSAGE_TEXT
| PARAMETER_MODE
| PARAMETER_NAME
| PARAMETER_ORDINAL_POSITION
| RETURNED_SQLSTATE
| ROUTINE_CATALOG
| ROUTINE_NAME
| ROUTINE_SCHEMA
| SCHEMA_NAME
| SERVER_NAME
| SPECIFIC_NAME
| SUBCLASS_ORIGIN
| TABLE_NAME
| TRIGGER_CATALOG
| TRIGGER_NAME
| TRIGGER_SCHEMA

<condition number> ::= <simple value specification>
```

Syntax Rules

1) The declared type of a <simple target specification> contained in a <statement information item> or <condition information item> shall be the data type specified in Table 30, "<identifier>s for use with <get diagnostics statement>", for the corresponding <statement information item name> or <condition information item name>.

2) The declared type of <condition number> shall be exact numeric with scale 0 (zero).

Table 30 — <identifier>s for use with <get diagnostics statement>

<identifier>	Declared Type
COMMAND_FUNCTION	variable-length character string with maximum length $L^{\dagger}$
COMMAND_FUNC-TION_CODE	exact numeric with scale 0 (zero)
DYNAMIC_FUNCTION	variable-length character string with maximum length $L^{\dagger}$
DYNAMIC_FUNCTION_CODE	exact numeric with scale 0 (zero)
MORE	fixed-length character string with length 1
NUMBER	exact numeric with scale 0 (zero)

<identifier>	Declared Type
ROW_COUNT	exact numeric with scale 0 (zero)
TRANSACTION_ACTIVE	exact numeric with scale 0 (zero)
TRANSACTIONS_COMMIT-TED	exact numeric with scale 0 (zero)
TRANSAC-TIONS_ROLLED_BACK	exact numeric with scale 0 (zero)
CATALOG_NAME	variable-length character string with maximum length *L*
CLASS_ORIGIN	variable-length character string with maximum length *L*
COLUMN_NAME	variable-length character string with maximum length *L*
CONDITION_NUMBER	exact numeric with scale 0 (zero)
CONNECTION_NAME	variable-length character string with maximum length *L*
CONSTRAINT_CATALOG	variable-length character string with maximum length *L*
CONSTRAINT_NAME	variable-length character string with maximum length *L*
CONSTRAINT_SCHEMA	variable-length character string with maximum length *L*
CURSOR_NAME	variable-length character string with maximum length *L*
MESSAGE_LENGTH	exact numeric with scale 0 (zero)
MESSAGE_OCTET_LENGTH	exact numeric with scale 0 (zero)
MESSAGE_TEXT	variable-length character string with maximum length *L*
PARAMETER_MODE	variable-length character string with maximum length 5
PARAMETER_NAME	variable-length character string with maximum length *L*
PARAMETER_ORDINAL_POSI-TION	exact numeric with scale 0 (zero)
RETURNED_SQLSTATE	fixed-length character string with length 5
ROUTINE_CATALOG	variable-length character string with maximum length *L*
ROUTINE_NAME	variable-length character string with maximum length *L*
ROUTINE_SCHEMA	variable-length character string with maximum length *L*

<identifier>	Declared Type
SCHEMA_NAME	variable-length character string with maximum length *L*
SERVER_NAME	variable-length character string with maximum length *L*
SPECIFIC_NAME	variable-length character string with maximum length *L*
SUBCLASS_ORIGIN	variable-length character string with maximum length *L*
TABLE_NAME	variable-length character string with maximum length *L*
TRIGGER_CATALOG	variable-length character string with maximum length *L*
TRIGGER_NAME	variable-length character string with maximum length *L*
TRIGGER_SCHEMA	variable-length character string with maximum length *L*
† **Where** *L* is an implementation-defined integer not less than 128.	

Access Rules

None.

General Rules

1) Let *DA* be the first diagnostics area.

2) Specification of <statement information item> assigns the value of the specified statement information item in *DA* to <simple target specification>.

   a) The value of NUMBER is the number of exception or completion conditions that have been stored in *DA* as a result of executing the previous SQL-statement other than a <get diagnostics statement>.

      NOTE 453 — The <get diagnostics statement> itself may return information via the SQLSTATE parameter, but does not modify the previous contents of *DA*.

   b) The value of MORE is:

Y	More conditions were raised during execution of the SQL-statement than there are condition areas in *DA*.
N	All of the conditions that were raised during execution of the SQL-statement have been stored in *DA*.

   c) The value of COMMAND_FUNCTION is the identification of the SQL-statement executed. Table 31, "SQL-statement codes" specifies the identifier of the SQL-statements.

d) The value of COMMAND_FUNCTION_CODE is a number identifying the SQL-statement executed. Table 31, "SQL-statement codes" specifies the code for the SQL-statements. Positive values are reserved for SQL-statements defined by ISO/IEC 9075; negative values are reserved for implementation-defined SQL-statements.

e) The value of DYNAMIC_FUNCTION is a character string that identifies the type of the SQL-statement being prepared or executed dynamically. Table 31, "SQL-statement codes", specifies the identifier of the SQL-statements.

f) The value of DYNAMIC_FUNCTION_CODE is a number that identifies the type of the SQL-statement being prepared or executed dynamically. Table 31, "SQL-statement codes", specifies the code for the SQL-statements. Positive values are reserved for SQL-statements defined by ISO/IEC 9075; negative values are reserved for implementation-defined SQL-statements.

Table 31 — SQL-statement codes

SQL-statement	Identifier	Code
<allocate cursor statement>	ALLOCATE CURSOR	1 (one)
<allocate descriptor statement>	ALLOCATE DESCRIPTOR	2
<alter domain statement>	ALTER DOMAIN	3
<alter routine statement>	ALTER ROUTINE	17
<alter sequence generator statement>	ALTER SEQUENCE	134
<alter type statement>	ALTER TYPE	60
<alter table statement>	ALTER TABLE	4
<alter transform statement>	ALTER TRANSFORM	127
<assertion definition>	CREATE ASSERTION	6
<call statement>	CALL	7
<character set definition>	CREATE CHARACTER SET	8
<close statement>	CLOSE CURSOR	9
<collation definition>	CREATE COLLATION	10
<commit statement>	COMMIT WORK	11
<connect statement>	CONNECT	13
<deallocate descriptor statement>	DEALLOCATE DESCRIPTOR	15

SQL-statement	Identifier	Code
<deallocate prepared statement>	DEALLOCATE PREPARE	16
<delete statement: positioned>	DELETE CURSOR	18
<delete statement: searched>	DELETE WHERE	19
<describe statement>	DESCRIBE	20
<direct select statement: multiple rows>	SELECT	21
<disconnect statement>	DISCONNECT	22
<domain definition>	CREATE DOMAIN	23
<drop assertion statement>	DROP ASSERTION	24
<drop character set statement>	DROP CHARACTER SET	25
<drop collation statement>	DROP COLLATION	26
<drop data type statement>	DROP TYPE	35
<drop domain statement>	DROP DOMAIN	27
<drop role statement>	DROP ROLE	29
<drop routine statement>	DROP ROUTINE	30
<drop schema statement>	DROP SCHEMA	31
<drop sequence generator statement>	DROP SEQUENCE	135
<drop table statement>	DROP TABLE	32
<drop transform statement>	DROP TRANSFORM	116
<drop transliteration statement>	DROP TRANSLATION	33
<drop trigger statement>	DROP TRIGGER	34
<drop user-defined cast statement>	DROP CAST	78
<drop user-defined ordering statement>	DROP ORDERING	115
<drop view statement>	DROP VIEW	36
<dynamic close statement>	DYNAMIC CLOSE	37
<dynamic delete statement: positioned>	DYNAMIC DELETE CURSOR	38

SQL-statement	Identifier	Code
<dynamic fetch statement>	DYNAMIC FETCH	39
<dynamic open statement>	DYNAMIC OPEN	40
<dynamic select statement>	SELECT CURSOR	85
<dynamic single row select statement>	SELECT	41
<dynamic update statement: positioned>	DYNAMIC UPDATE CURSOR	42
<execute immediate statement>	EXECUTE IMMEDIATE	43
<execute statement>	EXECUTE	44
<fetch statement>	FETCH	45
<free locator statement>	FREE LOCATOR	98
<get descriptor statement>	GET DESCRIPTOR	47
<hold locator statement>	HOLD LOCATOR	99
<grant privilege statement>	GRANT	48
<grant role statement>	GRANT ROLE	49
<insert statement>	INSERT	50
<merge statement>	MERGE	128
<open statement>	OPEN	53
<preparable dynamic delete statement: positioned>	PREPARABLE DYNAMIC DELETE CURSOR	54
<preparable dynamic update statement: positioned>	PREPARABLE DYNAMIC UPDATE CURSOR	55
<prepare statement>	PREPARE	56
<release savepoint statement>	RELEASE SAVEPOINT	57
<return statement>	RETURN	58
<revoke privilege statement>	REVOKE	59
<revoke role statement>	REVOKE ROLE	129
<role definition>	CREATE ROLE	61

SQL-statement	Identifier	Code
<rollback statement>	ROLLBACK WORK	62
<savepoint statement>	SAVEPOINT	63
<schema definition>	CREATE SCHEMA	64
<schema routine>	CREATE ROUTINE	14
<select statement: single row>	SELECT	65
<sequence generator definition>	CREATE SEQUENCE	133
<set catalog statement>	SET CATALOG	66
<set connection statement>	SET CONNECTION	67
<set constraints mode statement>	SET CONSTRAINT	68
<set descriptor statement>	SET DESCRIPTOR	70
<set local time zone statement>	SET TIME ZONE	71
<set names statement>	SET NAMES	72
<set path statement>	SET PATH	69
<set role statement>	SET ROLE	73
<set schema statement>	SET SCHEMA	74
<set session user identifier statement>	SET SESSION AUTHORIZATION	76
<set session characteristics statement>	SET SESSION CHARACTERISTICS	109
<set session collation statement>	SET COLLATION	136
<set transform group statement>	SET TRANSFORM GROUP	118
<set transaction statement>	SET TRANSACTION	75
<start transaction statement>	START TRANSACTION	111
<table definition>	CREATE TABLE	77
<transform definition>	CREATE TRANSFORM	117
<transliteration definition>	CREATE TRANSLATION	79
<trigger definition>	CREATE TRIGGER	80

SQL-statement	Identifier	Code
<update statement: positioned>	UPDATE CURSOR	81
<update statement: searched>	UPDATE WHERE	82
<user-defined cast definition>	CREATE CAST	52
<user-defined type definition>	CREATE TYPE	83
<user-defined ordering definition>	CREATE ORDERING	114
<view definition>	CREATE VIEW	84
Implementation-defined statements	An implementation-defined character string value different from the value associated with any other SQL-statement	*x*[1]
Unrecognized statements	A zero-length string	0 (zero)
[1] An implementation-defined negative number different from the value associated with any other SQL-statement.		

NOTE 454 — Other, additional, values are used in other parts of ISO/IEC 9075; please see the corresponding table in the other parts of ISO/IEC 9075; for more information.

g) The value of ROW_COUNT is the number of rows affected as the result of executing a <delete statement: searched>, <insert statement>, <merge statement>, or <update statement: searched> or as a direct result of executing the previous SQL-statement. Let *S* be the <delete statement: searched>, <insert statement>, <merge statement>, or <update statement: searched>. Let *T* be the table identified by the <table name> directly contained in *S*.

Case:

i) If *S* is an <insert statement>, then the value of ROW_COUNT is the number of rows inserted into *T*.

ii) If *S* is a <merge statement>, then let *TR1* be the <target table> immediately contained in *S*, let *TR2* be the <table reference> immediately contained in *S*, and let *SC* be the <search condition> immediately contained in *S*. If <merge correlation name> is specified, let *MCN* be "AS <merge correlation name>"; otherwise, let *MCN* be a zero-length string.

Case:

1) If *S* contains a <merge when matched clause> and does not contain a <merge when not matched clause>, then the value of ROW_COUNT is effectively derived by executing the statement:

```
SELECT COUNT (*)
FROM TR1 MCN, TR2
WHERE SC
```

before the execution of *S*.

2) If *S* contains a <merge when not matched clause> and does not contain a <merge when matched clause>, then the value of ROW_COUNT is effectively derived by executing the statement:

```
( SELECT COUNT(*)
 FROM TR1 MCN
   RIGHT OUTER JOIN
      TR2
     ON SC )
-
( SELECT COUNT (*)
  FROM TR1 MCN, TR2
  WHERE SC )
```

before the execution of *S*.

3) If *S* contains both a <merge when matched clause> and a <merge when not matched clause>, then the value of ROW_COUNT is effectively derived by executing the statement:

```
SELECT COUNT(*)
FROM TR1 MCN
  RIGHT OUTER JOIN
     TR2
    ON SC
```

before the execution of *S*.

iii) If <correlation name> is specified, then let *MCN* be "AS <correlation name>"; otherwise, let *MCN* be a zero-length string. If *S* is a <delete statement: searched> or an <update statement: searched>, then

Case:

1) If *S* does not contain a <search condition>, then the value of ROW_COUNT is the cardinality of *T* before the execution of *S*.

2) Otherwise, let *SC* be the <search condition> directly contained in *S*. The value of ROW_COUNT is effectively derived by executing the statement:

```
SELECT COUNT(*)
FROM T MCN
WHERE SC
```

before the execution of *S*.

The value of ROW_COUNT following the execution of an SQL-statement that does not directly result in the execution of a <delete statement: searched>, an <insert statement>, a <merge statement>, or an <update statement: searched> is implementation-dependent.

h) The value of TRANSACTIONS_COMMITTED is the number of SQL-transactions that have been committed since the most recent time at which *DA* was emptied.

NOTE 455 — See the General Rules of Subclause 13.3, "<externally-invoked procedure>", and Subclause 13.4, "Calls to an <externally-invoked procedure>". TRANSACTIONS_COMMITTED indicates the number of SQL-transactions that were committed during the invocation of an external routine.

i) The value of TRANSACTIONS_ROLLED_BACK is the number of SQL-transactions that have been rolled back since the most recent time at which *DA* was emptied.

NOTE 456 — See the General Rules of Subclause 13.3, "<externally-invoked procedure>", and Subclause 13.4, "Calls to an <externally-invoked procedure>". TRANSACTIONS_ROLLED_BACK indicates the number of SQL-transactions that were rolled back during the invocation of an external routine.

j) The value of TRANSACTION_ACTIVE is 1 (one) if an SQL-transaction is currently active, and 0 (zero) if an SQL-transaction is not currently active.

NOTE 457 — TRANSACTION_ACTIVE indicates whether an SQL-transaction is active upon return from an external routine.

k) It is implementation-defined whether the identifier and code from Table 31, "SQL-statement codes", for <dynamic select statement> or <dynamic single row select statement> is used to describe a <dynamic select statement> or <dynamic single row select statement> that has been prepared but has not yet been executed dynamically.

3) If <condition information> is specified, then let *N* be the value of <condition number>. If *N* is less than 1 (one) or greater than the number of occupied condition areas in *DA*, then an exception condition is raised: *invalid condition number*. If <condition number> has the value 1 (one), then the diagnostics information retrieved corresponds to the condition indicated by the SQLSTATE value actually returned by execution of the previous SQL-statement other than a <get diagnostics statement>. Otherwise, the association between <condition number> values and specific conditions raised during evaluation of the General Rules for that SQL-statement is implementation-dependent.

4) Specification of <condition information item> assigns the value of the specified condition information item in the *N*-th condition area in *DA* to <simple target specification>.

a) The value of CONDITION_NUMBER is the value of <condition number>.

b) The value of CLASS_ORIGIN is the identification of the naming authority that defined the class value of RETURNED_SQLSTATE. That value shall be 'ISO 9075' for any RETURNED_SQLSTATE whose class value is fully defined in Subclause 23.1, "SQLSTATE", and shall be an implementation-defined character string other than 'ISO 9075' for any RETURNED_SQLSTATE whose class value is an implementation-defined class value.

c) The value of SUBCLASS_ORIGIN is the identification of the naming authority that defined the subclass value of RETURNED_SQLSTATE. That value shall be 'ISO 9075' for any RETURNED_SQLSTATE whose subclass value is fully defined in Subclause 23.1, "SQLSTATE", and shall be an implementation-defined character string other than 'ISO 9075' for any RETURNED_SQLSTATE whose subclass value is an implementation-defined subclass value.

d) The value of RETURNED_SQLSTATE is the SQLSTATE parameter that would have been returned if this were the only completion or exception condition possible.

e) If the value of RETURNED_SQLSTATE corresponds to *warning* with a subclass of *cursor operation conflict*, then the value of CURSOR_NAME is the name of the cursor that caused the completion condition to be raised.

f) If the value of RETURNED_SQLSTATE corresponds to *integrity constraint violation*, *transaction rollback — integrity constraint violation*, or a *triggered data change violation* that was caused by a violation of a referential constraint, then:

i) The values of CONSTRAINT_CATALOG and CONSTRAINT_SCHEMA are the <catalog name> and the <unqualified schema name> of the <schema name> of the schema containing

the constraint or assertion. The value of CONSTRAINT_NAME is the <qualified identifier> of the constraint or assertion.

ii) Case:

1) If the violated integrity constraint is a table constraint, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name>, and the <qualified identifier>, respectively, of the table in which the table constraint is contained.

2) If the violated integrity constraint is an assertion and if only one table referenced by the assertion has been modified as a result of executing the SQL-statement, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name>, and the <qualified identifier>, respectively, of the modified table.

3) Otherwise, the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are a zero-length string.

If TABLE_NAME identifies a declared local temporary table, then CATALOG_NAME is a zero-length string and SCHEMA_NAME is "MODULE".

g) If the value of RETURNED_SQLSTATE corresponds to *triggered action exception*, *transaction rollback — triggered action exception*, or a *triggered data change violation* that was caused by a trigger, then:

i) The values of TRIGGER_CATALOG and TRIGGER_SCHEMA are the <catalog name> and the <unqualified schema name> of the <schema name> of the schema containing the trigger. The value of TRIGGER_NAME is the <qualified identifier> of the <trigger name> of the trigger.

ii) The values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name>, and the <qualified identifier> of the <table name>, respectively, of the table on which the trigger is defined.

h) If the value of RETURNED_SQLSTATE corresponds to *syntax error or access rule violation*, then:

i) Case:

1) If the syntax error or access rule violation was caused by reference to a specific table, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are

Case:

A) If the specific table referenced was not a declared local temporary table, then the <catalog name>, the <unqualified schema name> of the <schema name> of the schema that contains the table that caused the syntax error or access rule violation, and the <qualified identifier>, respectively.

B) Otherwise, the zero-length string, "MODULE", and the <qualified identifier>, respectively.

2) Otherwise, CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME contain a zero-length string.

ii) If the syntax error or access rule violation was for an inaccessible column, then the value of COLUMN_NAME is the <column name> of that column. Otherwise, the value of COLUMN_NAME is a zero-length string.

i) If the value of RETURNED_SQLSTATE corresponds to *invalid cursor state*, then the value of CURSOR_NAME is the name of the cursor that is in the invalid state.

j) If the value of RETURNED_SQLSTATE corresponds to *with check option violation*, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name> of the schema that contains the view that caused the violation of the WITH CHECK OPTION, and the <qualified identifier> of that view, respectively.

k) If the value of RETURNED_SQLSTATE does not correspond to *syntax error or access rule violation*, then:

i) If the values of CATALOG_NAME, SCHEMA_NAME, TABLE_NAME, and COLUMN_NAME identify a column for which no privileges are granted to the enabled authorization identifiers, then the value of COLUMN_NAME is replaced by a zero-length string.

ii) If the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME identify a table for which no privileges are granted to the enabled authorization identifiers, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are replaced by a zero-length string.

iii) If the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME identify a <table constraint> for some table *T* and if no privileges for *T* are granted to the enabled authorization identifiers, then the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME are replaced by a zero-length string.

iv) If the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME identify an assertion contained in some schema *S* and if the owner of *S* is not included in the set of enabled authorization identifiers, then the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME are replaced by a zero-length string.

l) If the value of RETURNED_SQLSTATE corresponds to *external routine invocation exception*, *external routine exception*, *SQL routine exception*, or *warning*, then

i) The values of ROUTINE_CATALOG and ROUTINE_SCHEMA are the <catalog name> and the <unqualified schema name>, respectively, of the <schema name> of the schema containing the SQL-invoked routine.

ii) The values of ROUTINE_NAME and SPECIFIC_NAME are the <identifier> of the <routine name> and the <identifier> of the <specific name> of the SQL-invoked routine, respectively.

iii) Case:

1) If the condition is related to parameter P_i of the SQL-invoked routine, then:

A) The value of PARAMETER_MODE is the <parameter mode> of P_i.

B) The value of PARAMETER_ORDINAL_POSITION is the value of *i*.

C) The value of PARAMETER_NAME is a zero-length string.

2) Otherwise:

A) The value of PARAMETER_MODE is a zero-length string.

B) The value of PARAMETER_ORDINAL_POSITION is 0 (zero).

C) The value of PARAMETER_NAME is a zero-length string.

m) If the value of RETURNED_SQLSTATE corresponds to *external routine invocation exception*, *external routine exception*, *SQL routine exception*, or *warning*, then the value of MESSAGE_TEXT is the message text item of the SQL-invoked routine that raised the exception. Otherwise the value of MESSAGE_TEXT is an implementation-defined character string.

NOTE 458 — An SQL-implementation may set this to <space>s, to a zero-length string, or to a character string describing the condition indicated by RETURNED_SQLSTATE.

n) The value of MESSAGE_LENGTH is the length in characters of the character string value in MESSAGE_TEXT.

o) The value of MESSAGE_OCTET_LENGTH is the length in octets of the character string value in MESSAGE_TEXT.

p) The values of CONNECTION_NAME and SERVER_NAME are respectively

Case:

i) If COMMAND_FUNCTION or DYNAMIC_FUNCTION identifies an <SQL connection statement>, then the <connection name> and the <SQL-server name> specified by or implied by the <SQL connection statement>.

ii) Otherwise, the <connection name> and <SQL-server name> of the SQL-session in which the condition was raised.

q) If the value of RETURNED_SQLSTATE corresponds to *data exception — numeric value out of range*, *data exception — invalid character value for cast*, *data exception — string data, right truncation*, *data exception — interval field overflow*, *integrity constraint violation*, or *warning — string data, right truncation*, and the condition was raised as the result of an assignment to an SQL parameter during an SQL-invoked routine invocation, then:

i) The values of ROUTINE_CATALOG and ROUTINE_SCHEMA are the <catalog name> and the <unqualified schema name>, respectively, of the <schema name> of the schema containing the routine.

ii) The values of the ROUTINE_NAME and SPECIFIC_NAME are the <identifier> of the <routine name> and the <identifier> of the <specific name>, respectively, of the routine.

iii) If the condition is related to parameter P_i of the SQL-invoked routine, then:

1) The value of PARAMETER_MODE is the <parameter mode> of P_i.

2) The value of PARAMETER_ORDINAL_POSITION is the value of i.

3) If an <SQL parameter name> was specified for the SQL parameter when the SQL-invoked routine was created, then the value of PARAMETER_NAME is the <SQL parameter name> of P_i. Otherwise, the value of PARAMETER_NAME is a zero-length string.

5) The values of character string items where not otherwise specified by the preceding rules are set to a zero-length string.

NOTE 459 — There are no numeric items that are not set by these rules.

6) The General Rules of Subclause 9.2, "Store assignment", apply to <simple target specification> and whichever of <statement information item name> or <condition information item name> is specified, as *TARGET* and *VALUE*, respectively.

Conformance Rules

1) Without Feature F121, "Basic diagnostics management", conforming SQL language shall not contain a <get diagnostics statement>.

2) Without Feature T511, "Transaction counts", conforming SQL language shall not contain a <statement information item name> that contains TRANSACTIONS_COMMITTED, TRANSACTIONS_ROLLED_BACK, or TRANSACTION_ACTIVE.

22.2 Pushing and popping the diagnostics area stack

Function

Define operations on the diagnostics area stack.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *OP* be the *OPERATION* and let *DAS* be the *STACK* specified in an application of this Subclause.

2) Case:

   a) If *OP* is "PUSH", then

      Case:

      i) If the number of diagnostics areas in *DAS* is equal to the implementation-dependent maximum number of diagnostics areas per diagnostics area stack, then an exception condition is raised: *diagnostics exception — maximum number of stacked diagnostics areas exceeded.*

      ii) Otherwise, *DAS* is pushed and the contents of the second diagnostics area in *DAS* are copied to the first.

   b) If *OP* is "POP", then the first diagnostics area is removed from *DAS* such that all subsequent diagnostics areas in *DAS* move up one position, the second becoming the first, the third becoming the second, and so on.

Conformance Rules

None.

23 Status codes

23.1 SQLSTATE

The character string value returned in an SQLSTATE parameter comprises a 2-character class value followed by a 3-character subclass value, each with an implementation-defined character set that has a one-octet character encoding form and is restricted to <digit>s and <simple Latin upper case letter>s. Table 32, "SQLSTATE class and subclass values", specifies the class value for each condition and the subclass value or values for each class value.

Class values that begin with one of the <digit>s '0', '1', '2', '3', or '4' or one of the <simple Latin upper case letter>s 'A', 'B', 'C', 'D', 'E', 'F', 'G', or 'H' are returned only for conditions defined in ISO/IEC 9075 or in any other International Standard. The range of such class values are called *standard-defined classes*. Some such class codes are reserved for use by specific International Standards, as specified elsewhere in this Clause. Subclass values associated with such classes that also begin with one of those 13 characters are returned only for conditions defined in ISO/IEC 9075 or some other International Standard. The range of such class values are called *standard-defined classes*. Subclass values associated with such classes that begin with one of the <digit>s '5', '6', '7', '8', or '9' or one of the <simple Latin upper case letter>s 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', or 'Z' are reserved for implementation-specified conditions and are called *implementation-defined subclasses*.

Class values that begin with one of the <digit>s '5', '6', '7', '8', or '9' or one of the <simple Latin upper case letter>s 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', or 'Z' are reserved for implementation-specified exception conditions and are called *implementation-defined classes*. All subclass values except '000', which means *no subclass*, associated with such classes are reserved for implementation-specified conditions and are called *implementation-defined subclasses*. An implementation-defined completion condition shall be indicated by returning an implementation-defined subclass in conjunction with one of the classes *successful completion*, *warning*, or *no data*.

If a subclass value is not specified for a condition, then either subclass '000' or an implementation-defined subclass is returned.

NOTE 460 — One consequence of this is that an SQL-implementation may, but is not required by ISO/IEC 9075 to, provide subcodes for exception condition *syntax error or access rule violation* that distinguish between the syntax error and access rule violation cases.

If multiple completion conditions: *warning* or multiple exception conditions, including implementation-defined exception conditions, are raised, then it is implementation-dependent which of the corresponding SQLSTATE values is returned in the SQLSTATE status parameter, provided that the precedence rules in Subclause 4.29.2, "Status parameters", are obeyed. Any number of applicable conditions values in addition to the one returned in the SQLSTATE status parameter, may be returned in the diagnostics area.

An implementation-specified condition may duplicate, in whole or in part, a condition defined in ISO/IEC 9075; however, if such a condition occurs as a result of executing a statement, then the corresponding implementation-defined SQLSTATE value shall not be returned in the SQLSTATE parameter but may be returned in the diagnostics area.

The "Category" column has the following meanings: "S" means that the class value given corresponds to successful completion and is a completion condition; "W" means that the class value given corresponds to a successful completion but with a warning and is a completion condition; "N" means that the class value given corresponds to a no-data situation and is a completion condition; "X" means that the class value given corresponds to an exception condition.

Table 32 — SQLSTATE class and subclass values

Category	Condition	Class	Subcondition	Subclass
X	*ambiguous cursor name*	3C	(*no subclass*)	000
X	*attempt to assign to non-updatable column*	0U	(*no subclass*)	000
X	*attempt to assign to ordering column*	0V	(*no subclass*)	000
X	*cardinality violation*	21	(*no subclass*)	000
X	*connection exception*	08	(*no subclass*)	000
			connection does not exist	003
			connection failure	006
			connection name in use	002
			SQL-client unable to establish SQL-connection	001
			SQL-server rejected establishment of SQL-connection	004
			transaction resolution unknown	007
X	*cursor sensitivity exception*	36	(*no subclass*)	000
			request failed	002
			request rejected	001
X	*data exception*	22	(*no subclass*)	000
			array data, right truncation	02F
			array element error	02E
			character not in repertoire	021

Category	Condition	Class	Subcondition	Subclass
			datetime field overflow	008
			division by zero	012
			error in assignment	005
			escape character conflict	00B
			indicator overflow	022
			interval field overflow	015
			interval value out of range	00P
			invalid argument for natural logarithm	01E
			invalid argument for power function	01F
			invalid argument for width bucket function	01G
			invalid character value for cast	018
			invalid datetime format	007
			invalid escape character	019
			invalid escape octet	00D
			invalid escape sequence	025
			invalid indicator parameter value	010
			invalid interval format	006
			invalid parameter value	023
			invalid preceding or following size in window function	013
			invalid regular expression	01B
			invalid repeat argument in a sample clause	02G
			invalid sample size	02H

Category	Condition	Class	Subcondition	Subclass
			invalid time zone displacement value	009
			invalid use of escape character	00C
			most specific type mismatch	00G
			multiset value overflow	00Q
			noncharacter in UCS string	029
			null value substituted for mutator subject parameter	02D
			null row not permitted in table	01C
			null value in array target	00E
			null value, no indicator parameter	002
			null value not allowed	004
			numeric value out of range	003
			sequence generator limit exceeded	00H
			string data, length mismatch	026
			string data, right truncation	001
			substring error	011
			trim error	027
			unterminated C string	024
			zero-length character string	00F
X	*dependent privilege descriptors still exist*	2B	(*no subclass*)	000
X	*diagnostics exception*	0Z	(*no subclass*)	000
			maximum number of stacked diagnostics areas exceeded	001
X	*dynamic SQL error*	07	(*no subclass*)	000

Category	Condition	Class	Subcondition	Subclass
			cursor specification cannot be executed	003
			data type transform function violation	00B
			invalid DATA target	00D
			invalid DATETIME_INTERVAL_CODE	00F
			invalid descriptor count	008
			invalid descriptor index	009
			invalid LEVEL value	00E
			prepared statement not a cursor specification	005
			restricted data type attribute violation	006
			undefined DATA value	00C
			using clause does not match dynamic parameter specifications	001
			using clause does not match target specifications	002
			using clause required for dynamic parameters	004
			using clause required for result fields	007
X	*external routine exception*	38	(*no subclass*)	000
			containing SQL not permitted	001
			modifying SQL-data not permitted	002
			prohibited SQL-statement attempted	003
			reading SQL-data not permitted	004

Category	Condition	Class	Subcondition	Subclass
X	*external routine invocation exception*	39	(*no subclass*)	000
			null value not allowed	004
X	*feature not supported*	0A	(*no subclass*)	000
			multiple server transactions	001
X	*integrity constraint violation*	23	(*no subclass*)	000
			restrict violation	001
X	*invalid authorization specification*	28	(*no subclass*)	000
X	*invalid catalog name*	3D	(*no subclass*)	000
X	*invalid character set name*	2C	(*no subclass*)	000
X	*invalid condition number*	35	(*no subclass*)	000
X	*invalid connection name*	2E	(*no subclass*)	000
X	*invalid cursor name*	34	(*no subclass*)	000
X	*invalid cursor state*	24	(*no subclass*)	000
X	*invalid grantor*	0L	(*no subclass*)	000
X	*invalid role specification*	0P	(*no subclass*)	000
X	*invalid schema name*	3F	(*no subclass*)	000
X	*invalid schema name list specification*	0E	(*no subclass*)	000
X	*invalid collation name*	2H	(*no subclass*)	000
X	*invalid SQL descriptor name*	33	(*no subclass*)	000
X	*invalid SQL-invoked procedure reference*	0M	(*no subclass*)	000
X	*invalid SQL statement name*	26	(*no subclass*)	000
X	*invalid SQL statement identifier*	30	(*no subclass*)	000
X	*invalid target type specification*	0D	(*no subclass*)	000

Category	Condition	Class	Subcondition	Subclass
X	*invalid transaction initiation*	0B	(*no subclass*)	000
X	*invalid transaction state*	25	(*no subclass*)	000
			active SQL-transaction	001
			branch transaction already active	002
			held cursor requires same isolation level	008
			inappropriate access mode for branch transaction	003
			inappropriate isolation level for branch transaction	004
			no active SQL-transaction for branch transaction	005
			read-only SQL-transaction	006
			schema and data statement mixing not supported	007
X	*invalid transaction termination*	2D	(*no subclass*)	000
X	*invalid transform group name specification*	0S	(*no subclass*)	000
X	*locator exception*	0F	(*no subclass*)	000
			invalid specification	001
N	*no data*	02	(*no subclass*)	000
			no additional dynamic result sets returned	001
X	*prohibited statement encountered during trigger execution*	0W	(*no subclass*)	000
X	Remote Database Access	*HZ*	(*See Table 33, "SQLSTATE class codes for RDA", for the definition of protocol subconditions and subclass code values*)	
X	*savepoint exception*	3B	(*no subclass*)	000

Category	Condition	Class	Subcondition	Subclass
			invalid specification	001
			too many	002
X	*SQL routine exception*	2F	(*no subclass*)	000
			function executed no return statement	005
			modifying SQL-data not permitted	002
			prohibited SQL-statement attempted	003
			reading SQL-data not permitted	004
S	*successful completion*	00	(*no subclass*)	000
X	*syntax error or access rule violation*	42	(*no subclass*)	000
X	*target table disagrees with cursor specification*	0T	(*no subclass*)	000
X	*transaction rollback*	40	(*no subclass*)	000
			integrity constraint violation	002
			serialization failure	001
			statement completion unknown	003
			triggered action exception	004
X	*triggered action exception*	09	(*no subclass*)	000
X	*triggered data change violation*	27	(*no subclass*)	000
W	*warning*	01	(*no subclass*)	000
			additional result sets returned	00D
			array data, right truncation	02F
			attempt to return too many result sets	00E
			cursor operation conflict	001

Category	Condition	Class	Subcondition	Subclass
			default value too long for information schema	00B
			disconnect error	002
			dynamic result sets returned	00C
			insufficient item descriptor areas	005
			null value eliminated in set function	003
			privilege not granted	007
			privilege not revoked	006
			query expression too long for information schema	00A
			search condition too long for information schema	009
			statement too long for information schema	00F
			string data, right truncation	004
X	*with check option violation*	44	(*no subclass*)	000

23.2 Remote Database Access SQLSTATE Subclasses

ISO/IEC 9075 reserves SQLSTATE class 'HZ' for Remote Database Access errors, which may occur when an SQL-client interacts with an SQL-server across a communications network using an RDA Application Context. [ISO9579], [ISO8649], and [ISO10026] define a number of exception conditions that shall be detected in a conforming ISO RDA implementation. This Subclause defines SQLSTATE subclass codes for each such condition out of the set of codes reserved for International Standards.

If an implementation using RDA reports a condition shown in Table 33, "SQLSTATE class codes for RDA", for a given exception condition, then it shall use the SQLSTATE class code 'HZ' and the subclass codes shown, and shall set the values of CLASS_ORIGIN to 'ISO 9075' and SUBCLASS_ORIGIN as indicated in Table 33, "SQLSTATE class codes for RDA", when those exceptions are retrieved by a <get diagnostics statement>.

An implementation using client-server communications other than RDA may report conditions corresponding to the conditions shown in Table 33, "SQLSTATE class codes for RDA", using the SQLSTATE class code 'HZ' and the corresponding subclass codes shown. It may set the values of CLASS_ORIGIN to 'ISO 9075' and SUBCLASS_ORIGIN as indicated in Table 33, "SQLSTATE class codes for RDA". Any other communications error shall be returned with a subclass code from the implementation-defined range, with CLASS_ORIGIN set to 'ISO 9075' and SUBCLASS_ORIGIN set to an implementation-defined character string.

A Remote Database Access exception may also result in an SQL completion condition defined in Table 32, "SQLSTATE class and subclass values" (such as '40000', *transaction rollback*); if such a condition occurs, then the 'HZ' class SQLSTATE shall not be returned in the SQLSTATE parameter, but may be returned in the Diagnostics Area.

Table 33 — SQLSTATE class codes for RDA

SQLSTATE Class	Subclass Origin
HZ	ISO/IEC 9579

24 Conformance

24.1 Claims of conformance to SQL/Foundation

In addition to the requirements of ISO/IEC 9075-1, in Clause 8, "Conformance", a claim of conformance to this part of ISO/IEC 9075 shall:

1) Claim conformance to at least one of:
   - Feature B011, "Embedded Ada"
   - Feature B012, "Embedded C"
   - Feature B013, "Embedded COBOL"
   - Feature B014, "Embedded Fortran"
   - Feature B015, "Embedded MUMPS"
   - Feature B016, "Embedded Pascal"
   - Feature B017, "Embedded PL/I"
   - Feature B111, "Module language Ada"
   - Feature B112, "Module language C"
   - Feature B113, "Module language COBOL"
   - Feature B114, "Module language Fortran"
   - Feature B115, "Module language MUMPS"
   - Feature B116, "Module language Pascal"
   - Feature B117, "Module language PL/I"
2) A claim conformance to at least one of:
   - Feature B121, "Routine language Ada"
   - Feature B122, "Routine language C"
   - Feature B123, "Routine language COBOL"
   - Feature B124, "Routine language Fortran"
   - Feature B125, "Routine language MUMPS"
   - Feature B126, "Routine language Pascal"
   - Feature B127, "Routine language PL/I"

— Feature B128, "Routine language SQL"

24.2 Additional conformance requirements for SQL/Foundation

An SQL-implementation that claims conformance to a feature in this part of ISO/IEC 9075 shall also claim conformance to the same feature, if present, in ISO/IEC 9075-11.

An SQL-implementation that claims conformance to Feature T061, "UCS support", shall:

— Conform to ISO/IEC 10646-1:2000 at some specified level.

— Provide at least one of the named character sets UTF8, UTF16, and UTF32.

— Provide, as the default collation for each such character set, a collation that conforms to ISO/IEC 14651:2001 at some level.

24.3 Implied feature relationships of SQL/Foundation

Table 34 — Implied feature relationships of SQL/Foundation

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
B032	Extended dynamic SQL	B031	Basic dynamic SQL
B034	Dynamic specification of cursor attributes	B031	Basic dynamic SQL
B111	Module language Ada	E182	Module language
B112	Module language C	E182	Module language
B113	Module language COBOL	E182	Module language
B114	Module language Fortran	E182	Module language
B115	Module language MUMPS	E182	Module language
B116	Module language Pascal	E182	Module language
B117	Module language PL/I	E182	Module language
F381	Extended schema manipulation	F491	Constraint management

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
F451	Character set definition	F461	Named character sets
F521	Assertions	F491	Constraint management
F691	Collation and translation	F695	Translation support
F691	Collation and translation	F690	Collation support
F693	SQL-session and client module collations	F690	Collation support
F711	ALTER domain	F251	Domain support
F721	Deferrable constraints	F491	Constraint management
F801	Full set function	F441	Extended set function support
S024	Enhanced structured types	S023	Basic structured types
S041	Basic reference types	S051	Create table of type
S043	Enhanced reference types	S041	Basic reference types
S051	Create table of type	S023	Basic structured types
S081	Subtables	S051	Create table of type
S092	Arrays of user-defined types	S091	Basic array support
S094	Arrays of reference types	S041	Basic reference types
S094	Arrays of reference types	S091	Basic array support
S095	Array constructors by query	S091	Basic array support
S096	Optional array bounds	S091	Basic array support
S111	ONLY in query expressions	S051	Create table of type
S201	SQL-invoked routines on arrays	S091	Basic array support
S202	SQL-invoked routines on multisets	S271	Basic multiset support
S231	Structured type locators	S023	Basic structured types
S232	Array locators	S091	Basic array support

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
S233	Multiset locators	S271	Basic multiset support
S242	Alter transform statement	S241	Transform functions
S272	Multisets of user-defined types	S271	Basic multiset support
S274	Multisets of reference types	S041	Basic reference types
S274	Multisets of reference types	S271	Basic multiset support
S275	Advanced multiset support	S271	Basic multiset support
T042	Extended LOB data type support	T041	Basic LOB data type support
T061	UCS Support	F461	Named character sets
T061	UCS support	F690	Collation support
T122	WITH (excluding RECURSIVE) in subquery	T121	WITH (excluding RECURSIVE) in query expression
T131	Recursive query	T121	WITH (excluding RECURSIVE) in query expression
T132	Recursive query in subquery	T122	WITH (excluding RECURSIVE) in subquery
T132	Recursive query in subquery	T131	Recursive query
T173	Extended LIKE clause in table definition	T171	LIKE clause in table definition
T212	Enhanced trigger capability	T211	Basic trigger capability
T332	Extended roles	T331	Basic roles
T511	Transaction counts	F121	Basic diagnostics management
T571	Array-returning external SQL-invoked functions	S201	SQL-invoked routines on arrays
T572	Multiset-returning external SQL-invoked functions	S202	SQL-invoked routines on multisets
T612	Advanced OLAP operations	T611	Elementary OLAP operations