ISO/IEC JTC 1/SC 32

Date: 2006-02-01

CD 9075-2:200x(E)

ISO/IEC JTC 1/SC 32/WG 3

The United States of America (ANSI)

Information technology — Database languages — SQL —

Part 2:
Foundation (SQL/Foundation)

Technologies de l'information — Langages de base de données — SQL —
Partie 2: Fondations (SQL/Fondations)

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

Copyright notice

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

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

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

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

Violaters may be prosecuted.

Contents

Page

Foreword. xix

Introduction. xx

1 Scope. 1

2 Normative references. 3

2.1 JTC1 standards. 3

2.2 Other international standards. 4

3 Definitions, notations, and conventions. 5

3.1 Definitions. 5

3.1.1 Definitions taken from ISO/IEC 10646. 5

3.1.2 Definitions taken from ISO/IEC 14651. 5

3.1.3 Definitions taken from Unicode. 5

3.1.4 Definitions taken from ISO 8601. 6

3.1.5 Definitions taken from Part 1. 6

3.1.6 Definitions provided in Part 2. 6

3.2 Notation. 10

3.3 Conventions. 10

3.3.1 Use of terms. 11

3.3.1.1 Other terms. 11

4 Concepts. 13

4.1 Data types. 13

4.1.1 General introduction to data types. 13

4.1.2 Naming of predefined types. 14

4.1.3 Non-predefined and non-SQL types. 15

4.1.4 Comparison and ordering. 16

4.2 Character strings. 17

4.2.1 Introduction to character strings. 18

4.2.2 Comparison of character strings. 18

4.2.3 Operations involving character strings. 19

4.2.3.1 Regular expression syntaxes. 19

4.2.3.2 Operators that operate on character strings and return character strings. 19

4.2.3.3 Other operators involving character strings. 22

4.2.3.4 Operations involving large object character strings. 23

4.2.4 Character repertoires. 24

4.2.5 Character encoding forms. 24

4.2.6 Collations. 25

4.2.7 Character sets. . . . 26
4.2.8 Universal character sets. . . . 28
4.3 Binary strings. . . . 28
4.3.1 Introduction to binary strings. . . . 28
4.3.2 Binary string comparison. . . . 28
4.3.3 Operations involving binary strings. . . . 29
4.3.3.1 Operators that operate on binary strings and return binary strings. . . . 29
4.3.3.2 Other operators involving binary strings. . . . 29
4.4 Numbers. . . . 30
4.4.1 Introduction to numbers. . . . 30
4.4.2 Characteristics of numbers. . . . 30
4.4.3 Operations involving numbers. . . . 31
4.5 Boolean types. . . . 32
4.5.1 Introduction to Boolean types. . . . 32
4.5.2 Comparison and assignment of booleans. . . . 33
4.5.3 Operations involving booleans. . . . 33
4.5.3.1 Operations on booleans that return booleans. . . . 33
4.5.3.2 Other operators involving booleans. . . . 33
4.6 Datetimes and intervals. . . . 33
4.6.1 Introduction to datetimes and intervals. . . . 33
4.6.2 Datetimes. . . . 34
4.6.3 Intervals. . . . 37
4.6.4 Operations involving datetimes and intervals. . . . 38
4.7 User-defined types. . . . 39
4.7.1 Introduction to user-defined types. . . . 39
4.7.2 User-defined type descriptor. . . . 41
4.7.3 Observers and mutators. . . . 42
4.7.4 Constructors. . . . 42
4.7.5 Subtypes and supertypes. . . . 43
4.7.6 User-defined type comparison and assignment. . . . 44
4.7.7 Transforms for user-defined types. . . . 45
4.8 Row types. . . . 45
4.9 Reference types. . . . 46
4.9.1 Introduction to reference types. . . . 46
4.9.2 Operations involving references. . . . 47
4.10 Collection types. . . . 47
4.10.1 Introduction to collection types. . . . 47
4.10.2 Arrays. . . . 48
4.10.3 Multisets. . . . 48
4.10.4 Collection comparison and assignment. . . . 49
4.10.5 Operations involving arrays. . . . 49
4.10.5.1 Operators that operate on array values and return array elements. . . . 49
4.10.5.2 Operators that operate on array values and return array values. . . . 49
4.10.6 Operations involving multisets. . . . 49

4.10.6.1 Operators that operate on multisets and return multiset elements. . . . 49
4.10.6.2 Operators that operate on multisets and return multisets. . . . 50
4.11 Data conversions. . . . 50
4.12 Domains. . . . 51
4.13 Columns, fields, and attributes. . . . 51
4.14 Tables. . . . 53
4.14.1 Introduction to tables. . . . 53
4.14.2 Types of tables. . . . 53
4.14.3 Table descriptors. . . . 55
4.14.4 Relationships between tables. . . . 56
4.14.5 Referenceable tables, subtables, and supertables. . . . 57
4.14.6 Operations involving tables. . . . 58
4.14.7 Identity columns. . . . 59
4.14.8 Base columns and generated columns. . . . 59
4.14.9 Windowed tables. . . . 60
4.15 Data analysis operations (involving tables). . . . 61
4.15.1 Introduction to data analysis operations. . . . 61
4.15.2 Group functions. . . . 62
4.15.3 Window functions. . . . 62
4.15.4 Aggregate functions. . . . 63
4.16 Determinism. . . . 65
4.17 Integrity constraints. . . . 66
4.17.1 Overview of integrity constraints. . . . 66
4.17.2 Checking of constraints. . . . 66
4.17.3 Table constraints. . . . 67
4.17.4 Domain constraints. . . . 68
4.17.5 Assertions. . . . 69
4.18 Functional dependencies. . . . 69
4.18.1 Overview of functional dependency rules and notations. . . . 69
4.18.2 General rules and definitions. . . . 70
4.18.3 Known functional dependencies in a base table. . . . 71
4.18.4 Known functional dependencies in a transition table. . . . 71
4.18.5 Known functional dependencies in <table value constructor>. . . . 72
4.18.6 Known functional dependencies in a <joined table>. . . . 72
4.18.7 Known functional dependencies in a <table primary>. . . . 74
4.18.8 Known functional dependencies in a <table factor>. . . . 74
4.18.9 Known functional dependencies in a <table reference>. . . . 74
4.18.10 Known functional dependencies in the result of a <from clause>. . . . 75
4.18.11 Known functional dependencies in the result of a <where clause>. . . . 75
4.18.12 Known functional dependencies in the result of a <group by clause>. . . . 75
4.18.13 Known functional dependencies in the result of a <having clause>. . . . 76
4.18.14 Known functional dependencies in a <query specification>. . . . 76
4.18.15 Known functional dependencies in a <query expression>. . . . 77
4.19 Candidate keys. . . . 78

4.20 SQL-schemas. 78
4.21 Sequence generators. 80
4.21.1 General description of sequence generators. 80
4.21.2 Operations involving sequence generators. 81
4.22 SQL-client modules. 81
4.23 Embedded syntax. 82
4.24 Dynamic SQL concepts. 83
4.24.1 Overview of dynamic SQL. 83
4.24.2 Dynamic SQL statements and descriptor areas. 84
4.25 Direct invocation of SQL. 85
4.26 Externally-invoked procedures. 85
4.27 SQL-invoked routines. 85
4.27.1 Overview of SQL-invoked routines. 86
4.27.2 Characteristics of SQL-invoked routines. 87
4.27.3 Execution of SQL-invoked routines. 89
4.27.4 Routine descriptors. 89
4.27.5 Result sets returned by SQL-invoked procedures. 91
4.28 SQL-paths. 92
4.29 Host parameters. 93
4.29.1 Overview of host parameters. 93
4.29.2 Status parameters. 93
4.29.3 Data parameters. 94
4.29.4 Indicator parameters. 94
4.29.5 Locators. 94
4.30 Diagnostics area. 95
4.31 Standard programming languages. 96
4.32 Cursors. 97
4.32.1 General description of cursors. 97
4.32.2 Operations on and using cursors. 100
4.33 SQL-statements. 102
4.33.1 Classes of SQL-statements. 102
4.33.2 SQL-statements classified by function. 103
4.33.2.1 SQL-schema statements. 103
4.33.2.2 SQL-data statements. 105
4.33.2.3 SQL-data change statements. 105
4.33.2.4 SQL-transaction statements. 106
4.33.2.5 SQL-connection statements. 106
4.33.2.6 SQL-control statements. 106
4.33.2.7 SQL-session statements. 106
4.33.2.8 SQL-diagnostics statements. 107
4.33.2.9 SQL-dynamic statements. 107
4.33.2.10 SQL embedded exception declaration. 108
4.33.3 SQL-statements and SQL-data access indication. 108
4.33.4 SQL-statements and transaction states. 108

4.33.5 SQL-statement atomicity and statement execution contexts. . . . 110
4.33.6 Embeddable SQL-statements. . . . 111
4.33.7 Preparable and immediately executable SQL-statements. . . . 113
4.33.8 Directly executable SQL-statements. . . . 114
4.34 Basic security model. . . . 116
4.34.1 Authorization identifiers. . . . 116
4.34.1.1 SQL-session authorization identifiers. . . . 116
4.34.1.2 SQL-client module authorization identifiers. . . . 117
4.34.1.3 SQL-schema authorization identifiers. . . . 117
4.34.2 Privileges. . . . 117
4.34.3 Roles. . . . 120
4.34.4 Security model definitions. . . . 120
4.35 SQL-transactions. . . . 121
4.35.1 General description of SQL-transactions. . . . 121
4.35.2 Savepoints. . . . 121
4.35.3 Properties of SQL-transactions. . . . 122
4.35.4 Isolation levels of SQL-transactions. . . . 122
4.35.5 Implicit rollbacks. . . . 124
4.35.6 Effects of SQL-statements in an SQL-transaction. . . . 124
4.35.7 Encompassing transactions. . . . 125
4.36 SQL-connections. . . . 126
4.37 SQL-sessions. . . . 127
4.37.1 General description of SQL-sessions. . . . 127
4.37.2 SQL-session identification. . . . 127
4.37.3 SQL-session properties. . . . 128
4.37.4 Execution contexts. . . . 130
4.37.5 Routine execution context. . . . 131
4.38 Triggers. . . . 131
4.38.1 General description of triggers. . . . 131
4.38.2 Trigger execution. . . . 133
4.39 Client-server operation. . . . 135

5 Lexical elements. . . . 137

5.1 <SQL terminal character>. . . . 137
5.2 <token> and <separator>. . . . 141
5.3 <literal>. . . . 150
5.4 Names and identifiers. . . . 159

6 Scalar expressions. . . . 171

6.1 <data type>. . . . 171
6.2 <field definition>. . . . 183
6.3 <value expression primary>. . . . 185
6.4 <value specification> and <target specification>. . . . 187
6.5 <contextually typed value specification>. . . . 193
6.6 <identifier chain>. . . . 195

6.7 <column reference>. 199
6.8 <SQL parameter reference>. 201
6.9 <set function specification>. 202
6.10 <window function>. 204
6.11 <case expression>. 208
6.12 <cast specification>. 212
6.13 <next value expression>. 227
6.14 <field reference>. 229
6.15 <subtype treatment>. 230
6.16 <method invocation>. 232
6.17 <static method invocation>. 234
6.18 <new specification>. 236
6.19 <attribute or method reference>. 238
6.20 <dereference operation>. 240
6.21 <method reference>. 241
6.22 <reference resolution>. 243
6.23 <array element reference>. 245
6.24 <multiset element reference>. 246
6.25 <value expression>. 247
6.26 <numeric value expression>. 251
6.27 <numeric value function>. 253
6.28 <string value expression>. 265
6.29 <string value function>. 270
6.30 <datetime value expression>. 285
6.31 <datetime value function>. 288
6.32 <interval value expression>. 290
6.33 <interval value function>. 295
6.34 <boolean value expression>. 296
6.35 <array value expression>. 300
6.36 <array value constructor>. 302
6.37 <multiset value expression>. 304
6.38 <multiset value function>. 307
6.39 <multiset value constructor>. 308

7 Query expressions. 311

7.1 <row value constructor>. 311
7.2 <row value expression>. 314
7.3 <table value constructor>. 316
7.4 <table expression>. 318
7.5 <from clause>. 319
7.6 <table reference>. 321
7.7 <joined table>. 330
7.8 <where clause>. 339
7.9 <group by clause>. 340

7.10 <having clause>. 349
7.11 <window clause>. 351
7.12 <query specification>. 361
7.13 <query expression>. 371
7.14 <search or cycle clause>. 385
7.15 <subquery>. 390

8 Predicates. 393

8.1 <predicate>. 393
8.2 <comparison predicate>. 395
8.3 <between predicate>. 402
8.4 <in predicate>. 403
8.5 <like predicate>. 405
8.6 <similar predicate>. 411
8.7 <regex like predicate>. 417
8.8 <null predicate>. 419
8.9 <quantified comparison predicate>. 421
8.10 <exists predicate>. 423
8.11 <unique predicate>. 424
8.12 <normalized predicate>. 425
8.13 <match predicate>. 427
8.14 <overlaps predicate>. 430
8.15 <distinct predicate>. 432
8.16 <member predicate>. 434
8.17 <submultiset predicate>. 436
8.18 <set predicate>. 438
8.19 <type predicate>. 439
8.20 <search condition>. 441

9 Additional common rules. 443

9.1 Retrieval assignment. 443
9.2 Store assignment. 448
9.3 Result of data type combinations. 453
9.4 Subject routine determination. 456
9.5 Type precedence list determination. 457
9.6 Host parameter mode determination. 460
9.7 Type name determination. 462
9.8 Determination of identical values. 464
9.9 Equality operations. 466
9.10 Grouping operations. 469
9.11 Multiset element grouping operations. 471
9.12 Ordering operations. 473
9.13 Collation determination. 475
9.14 Execution of array-returning functions. 476
9.15 Execution of multiset-returning functions. 479

9.16 XQuery regular expression matching. . . . 480
9.17 XQuery regular expression replacement. . . . 483
9.18 Data type identity. . . . 485
9.19 Determination of a from-sql function. . . . 487
9.20 Determination of a from-sql function for an overriding method. . . . 488
9.21 Determination of a to-sql function. . . . 489
9.22 Determination of a to-sql function for an overriding method. . . . 490
9.23 Generation of the next value of a sequence generator. . . . 491
9.24 Creation of a sequence generator. . . . 492
9.25 Altering a sequence generator. . . . 494

10 Additional common elements. . . . 497

10.1 <interval qualifier>. . . . 497
10.2 <language clause>. . . . 501
10.3 <path specification>. . . . 503
10.4 <routine invocation>. . . . 504
10.5 <character set specification>. . . . 527
10.6 <specific routine designator>. . . . 529
10.7 <collate clause>. . . . 532
10.8 <constraint name definition> and <constraint characteristics>. . . . 533
10.9 <aggregate function>. . . . 535
10.10 <sort specification list>. . . . 546
10.11 Determination of view and view component privileges. . . . 548

11 Schema definition and manipulation. . . . 553

11.1 <schema definition>. . . . 553
11.2 <drop schema statement>. . . . 556
11.3 <table definition>. . . . 559
11.4 <column definition>. . . . 570
11.5 <default clause>. . . . 575
11.6 <table constraint definition>. . . . 579
11.7 <unique constraint definition>. . . . 581
11.8 <referential constraint definition>. . . . 583
11.9 <check constraint definition>. . . . 592
11.10 <alter table statement>. . . . 594
11.11 <add column definition>. . . . 595
11.12 <alter column definition>. . . . 597
11.13 <set column default clause>. . . . 598
11.14 <drop column default clause>. . . . 599
11.15 <add column scope clause>. . . . 600
11.16 <drop column scope clause>. . . . 601
11.17 <alter identity column specification>. . . . 603
11.18 <drop column definition>. . . . 604
11.19 <add table constraint definition>. . . . 606
11.20 <drop table constraint definition>. . . . 607

11.21 <drop table statement>. 610
11.22 <view definition>. 613
11.23 <drop view statement>. 622
11.24 <domain definition>. 625
11.25 <alter domain statement>. 628
11.26 <set domain default clause>. 629
11.27 <drop domain default clause>. 630
11.28 <add domain constraint definition>. 631
11.29 <drop domain constraint definition>. 632
11.30 <drop domain statement>. 633
11.31 <character set definition>. 635
11.32 <drop character set statement>. 637
11.33 <collation definition>. 639
11.34 <drop collation statement>. 641
11.35 <transliteration definition>. 643
11.36 <drop transliteration statement>. 646
11.37 <assertion definition>. 648
11.38 <drop assertion statement>. 650
11.39 <trigger definition>. 653
11.40 <drop trigger statement>. 657
11.41 <user-defined type definition>. 658
11.42 <attribute definition>. 674
11.43 <alter type statement>. 676
11.44 <add attribute definition>. 677
11.45 <drop attribute definition>. 679
11.46 <add original method specification>. 681
11.47 <add overriding method specification>. 687
11.48 <drop method specification>. 692
11.49 <drop data type statement>. 696
11.50 <SQL-invoked routine>. 699
11.51 <alter routine statement>. 724
11.52 <drop routine statement>. 727
11.53 <user-defined cast definition>. 729
11.54 <drop user-defined cast statement>. 731
11.55 <user-defined ordering definition>. 733
11.56 <drop user-defined ordering statement>. 736
11.57 <transform definition>. 738
11.58 <alter transform statement>. 741
11.59 <add transform element list>. 742
11.60 <drop transform element list>. 744
11.61 <drop transform statement>. 746
11.62 <sequence generator definition>. 749
11.63 <alter sequence generator statement>. 751
11.64 <drop sequence generator statement>. 752

12 Access control. 753
12.1 <grant statement>. 753
12.2 <grant privilege statement>. 758
12.3 <privileges>. 761
12.4 <role definition>. 765
12.5 <grant role statement>. 766
12.6 <drop role statement>. 768
12.7 <revoke statement>. 769
12.8 Grantor determination. 787
13 SQL-client modules. 789
13.1 <SQL-client module definition>. 789
13.2 <module name clause>. 794
13.3 <externally-invoked procedure>. 795
13.4 Calls to an <externally-invoked procedure>. 798
13.5 <SQL procedure statement>. 815
13.6 Data type correspondences. 822
14 Data manipulation. 833
14.1 <declare cursor>. 833
14.2 <cursor properties>. 835
14.3 <cursor specification>. 837
14.4 <open statement>. 842
14.5 <fetch statement>. 843
14.6 <close statement>. 847
14.7 <select statement: single row>. 848
14.8 <delete statement: positioned>. 852
14.9 <delete statement: searched>. 854
14.10 <insert statement>. 857
14.11 <merge statement>. 862
14.12 <update statement: positioned>. 869
14.13 <update statement: searched>. 871
14.14 <set clause list>. 875
14.15 <temporary table declaration>. 880
14.16 <free locator statement>. 882
14.17 <hold locator statement>. 883
15 Additional data manipulation rules. 885
15.1 Effect of opening a cursor. 885
15.2 Effect of advancing a received cursor to the next result set. 888
15.3 Determination of the current row of a cursor. 889
15.4 Effect of closing a cursor. 891
15.5 Effect of a positioned delete. 892
15.6 Effect of a positioned update. 894
15.7 Effect of deleting rows from base tables. 896
15.8 Effect of deleting some rows from a derived table. 898

15.9 Effect of deleting some rows from a viewed table. 900
15.10 Effect of inserting tables into base tables. 901
15.11 Effect of inserting a table into a derived table. 903
15.12 Effect of inserting a table into a viewed table. 905
15.13 Effect of replacing rows in base tables. 907
15.14 Effect of replacing some rows in a derived table. 909
15.15 Effect of replacing some rows in a viewed table. 912
15.16 Execution of BEFORE triggers. 914
15.17 Execution of AFTER triggers. 915
15.18 Execution of triggers. 916

16 Control statements. 919

16.1 <call statement>. 919
16.2 <return statement>. 920

17 Transaction management. 921

17.1 <start transaction statement>. 921
17.2 <set transaction statement>. 923
17.3 <transaction characteristics>. 925
17.4 <set constraints mode statement>. 927
17.5 <savepoint statement>. 929
17.6 <release savepoint statement>. 930
17.7 <commit statement>. 931
17.8 <rollback statement>. 933

18 Connection management. 935

18.1 <connect statement>. 935
18.2 <set connection statement>. 938
18.3 <disconnect statement>. 940

19 Session management. 943

19.1 <set session characteristics statement>. 943
19.2 <set session user identifier statement>. 945
19.3 <set role statement>. 946
19.4 <set local time zone statement>. 947
19.5 <set catalog statement>. 948
19.6 <set schema statement>. 949
19.7 <set names statement>. 951
19.8 <set path statement>. 952
19.9 <set transform group statement>. 953
19.10 <set session collation statement>. 954

20 Dynamic SQL. 957

20.1 Description of SQL descriptor areas. 957
20.2 <allocate descriptor statement>. 967
20.3 <deallocate descriptor statement>. 969
20.4 <get descriptor statement>. 970

20.5 <set descriptor statement>. 973
20.6 <prepare statement>. 977
20.7 <cursor attributes>. 988
20.8 <deallocate prepared statement>. 989
20.9 <describe statement>. 991
20.10 <input using clause>. 997
20.11 <output using clause>. 1001
20.12 <execute statement>. 1006
20.13 <execute immediate statement>. 1008
20.14 <dynamic declare cursor>. 1009
20.15 <allocate cursor statement>. 1011
20.16 <dynamic open statement>. 1014
20.17 <dynamic fetch statement>. 1016
20.18 <dynamic single row select statement>. 1017
20.19 <dynamic close statement>. 1018
20.20 <dynamic delete statement: positioned>. 1019
20.21 <dynamic update statement: positioned>. 1021
20.22 <preparable dynamic delete statement: positioned>. 1023
20.23 <preparable dynamic cursor name>. 1025
20.24 <preparable dynamic update statement: positioned>. 1027

21 Embedded SQL. 1029

21.1 <embedded SQL host program>. 1029
21.2 <embedded exception declaration>. 1041
21.3 <embedded SQL Ada program>. 1045
21.4 <embedded SQL C program>. 1052
21.5 <embedded SQL COBOL program>. 1060
21.6 <embedded SQL Fortran program>. 1066
21.7 <embedded SQL MUMPS program>. 1072
21.8 <embedded SQL Pascal program>. 1077
21.9 <embedded SQL PL/I program>. 1083

22 Direct invocation of SQL. 1089

22.1 <direct SQL statement>. 1089
22.2 <direct select statement: multiple rows>. 1093

23 Diagnostics management. 1095

23.1 <get diagnostics statement>. 1095
23.2 Pushing and popping the diagnostics area stack. 1112

24 Status codes. 1113

24.1 SQLSTATE. 1113
24.2 Remote Database Access SQLSTATE Subclasses. 1122

25 Conformance. 1123

25.1 Claims of conformance to SQL/Foundation. 1123
25.2 Additional conformance requirements for SQL/Foundation. 1124

25.3 Implied feature relationships of SQL/Foundation. 1124
Annex A SQL Conformance Summary. 1129
Annex B Implementation-defined elements. 1191
Annex C Implementation-dependent elements. 1209
Annex D Deprecated features. 1217
Annex E Incompatibilities with ISO/IEC 9075-2:2003. 1219
Annex F SQL feature taxonomy. 1221
Annex G Defect Reports not addressed in this edition of ISO/IEC 9075. 1251
Bibliography. 1255
Index. 1257

Tables

Table

Page

1 Overview of character sets. . . . 27
2 Fields in datetime values. . . . 34
3 Datetime data type conversions. . . . 36
4 Fields in year-month INTERVAL values. . . . 37
5 Fields in day-time INTERVAL values. . . . 37
6 Valid values for fields in INTERVAL values. . . . 38
7 Valid operators involving datetimes and intervals. . . . 39
8 SQL-transaction isolation levels and the three phenomena. . . . 123
9 Valid values for datetime fields. . . . 178
10 Valid absolute values for interval fields. . . . 179
11 Truth table for the AND boolean operator. . . . 299
12 Truth table for the OR boolean operator. . . . 299
13 Truth table for the IS boolean operator. . . . 299
14 <null predicate> semantics. . . . 420
15 Standard programming languages. . . . 501
16 Data type correspondences for Ada. . . . 822
17 Data type correspondences for C. . . . 823
18 Data type correspondences for COBOL. . . . 825
19 Data type correspondences for Fortran. . . . 826
20 Data type correspondences for M. . . . 828
21 Data type correspondences for Pascal. . . . 829
22 Data type correspondences for PL/I. . . . 830
23 Data types of <key word>s used in the header of SQL descriptor areas. . . . 961
24 Data types of <key word>s used in SQL item descriptor areas. . . . 961
25 Codes used for SQL data types in Dynamic SQL. . . . 963
26 Codes associated with datetime data types in Dynamic SQL. . . . 965
27 Codes used for <interval qualifier>s in Dynamic SQL. . . . 965
28 Codes used for input/output SQL parameter modes in Dynamic SQL. . . . 966
29 Codes associated with user-defined types in Dynamic SQL. . . . 966
30 <statement information item name>s for use with <get diagnostics statement>. . . . 1097
31 <condition information item name>s for use with <get diagnostics statement>. . . . 1097
32 SQL-statement codes. . . . 1100
33 SQLSTATE class and subclass values. . . . 1114
34 SQLSTATE class codes for RDA. . . . 1122
35 Implied feature relationships of SQL/Foundation. . . . 1124
36 Feature taxonomy and definition for mandatory features. . . . 1221
37 Feature taxonomy for optional features. . . . 1237

Figures

Figure

Page

1 Operation of <regular expression substring function>. 21
2 Illustration of WIDTH_BUCKET Semantics. 32

(Blank page)

Foreword

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

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

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

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

International Standard ISO/IEC 9075-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

This sixth edition cancels and replaces the fifth edition (ISO/IEC 9075-2:2003).

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

— Part 1: Framework (SQL/Framework)

— Part 2: Foundation (SQL/Foundation)

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

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

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

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

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

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

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

Introduction

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

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

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

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

4) Clause 4, “Concepts”, presents concepts used in the definition of SQL.

5) Clause 5, “Lexical elements”, defines the lexical elements of the language.

6) Clause 6, “Scalar expressions”, defines the elements of the language that produce scalar values.

7) Clause 7, “Query expressions”, defines the elements of the language that produce rows and tables of data.

8) Clause 8, “Predicates”, defines the predicates of the language.

9) Clause 9, “Additional common rules”, specifies the rules for assignments that retrieve data from or store data into SQL-data, and formation rules for set operations.

10) Clause 10, “Additional common elements”, defines additional language elements that are used in various parts of the language.

11) Clause 11, “Schema definition and manipulation”, defines facilities for creating and managing a schema.

12) Clause 12, “Access control”, defines facilities for controlling access to SQL-data.

13) Clause 13, “SQL-client modules”, defines SQL-client modules and externally-invoked procedures.

14) Clause 14, “Data manipulation”, defines the data manipulation statements.

15) Clause 15, “Additional data manipulation rules”, defines additional rules for data manipulation.

16) Clause 16, “Control statements”, defines the SQL-control statements.

17) Clause 17, “Transaction management”, defines the SQL-transaction management statements.

18) Clause 18, “Connection management” defines the SQL-connection management statements.

19) Clause 19, “Session management”, defines the SQL-session management statements.

20) Clause 20, “Dynamic SQL”, defines the SQL dynamic statements.

21) Clause 21, “Embedded SQL”, defines the host language embeddings.

22) Clause 22, “Direct invocation of SQL”, defines direct invocation of SQL language.

23) Clause 23, “Diagnostics management”, defines the diagnostics management facilities.

24) Clause 24, “Status codes”, defines values that identify the status of the execution of SQL-statements and the mechanisms by which those values are returned.

25) Clause 25, “Conformance”, defines the criteria for conformance to this part of ISO/IEC 9075.

26) Annex A, “SQL Conformance Summary”, is an informative Annex. It summarizes the conformance requirements of the SQL language.

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

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

29) Annex D, “Deprecated features”, is an informative Annex. It lists features that the responsible Technical Committee intend will not appear in a future revised version of this part of ISO/IEC 9075.

30) Annex E, “Incompatibilities with ISO/IEC 9075-2:2003”, is an informative Annex. It lists incompatibilities with the previous version of this part of ISO/IEC 9075.

31) Annex F, “SQL feature taxonomy”, is an informative Annex. It identifies features of the SQL language specified in this part of ISO/IEC 9075 by an identifier and a short descriptive name. This taxonomy is used to specify conformance and may be used to develop other profiles involving the SQL language.

32) Annex G, “Defect Reports not addressed in this edition of ISO/IEC 9075”, is an informative Annex. It describes the Defect Reports that were known at the time of publication of this part of this International Standard. Each of these problems is a problem carried forward from the previous edition of ISO/IEC 9075-2. No new problems have been created in the drafting of this edition of this International Standard.

In the text of this part of ISO/IEC 9075, Clauses begin a new odd-numbered page, and in Clause 5, “Lexical elements”, through Clause 24, “Status codes”, Subclauses begin a new page. Any resulting blank space is not significant.

(Blank page)

Information technology — Database languages — SQL —

Part 2:
Foundation (SQL/Foundation)

1 Scope

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

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

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

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

— For declaring database language procedures.

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

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

— For direct invocation of SQL-statements.

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

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

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

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

(Blank page)

2 Normative references

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

2.1 JTC1 standards

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

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

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

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

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

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

[ISO8601] ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

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

[ISO8649Amd1] ISO/IEC 8649:1996/Amd.1:1997, *Support of authentication mechanisms for the connectionless mode*.

[ISO8649Amd2] ISO/IEC 8649:1996/Amd.2:1998, *Fast-associate mechanism*.

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

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

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

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

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

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

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

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

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

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

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

[UCS] ISO/IEC 10646:2003, *Information technology — Universal Multi-Octet Coded Character Set (UCS)*.

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

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

ISO/IEC 14651:2001/Amd 1:2003, *AMENDMENT 1*.

ISO/IEC 14651:2001/Amd 2:2005, *AMENDMENT 2*.

2.2 Other international standards

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

[Schema2] (Recommendation) XML Schema Part 2: Datatypes, 28 October 2004, World Wide Web Consortium.
http://www/w3/org/TR/2004/REC-xmlschema-2-20041028

[Unicode 4.0.1] The Unicode Consortium, *The Unicode Standard, Version 4.0*, Reading, MA, Addison-Wesley Developers Press, 2003. ISBN 0-321-18578-1. As amended by *Unicode 4.0.1*.
http://www.unicode.org/versions/Unicode4.0.1

[Unicode10] Davis, Mark and Whistler, Ken. *Unicode Technical Standard #10, Unicode Collation Algorithm, Version 4.1.0*, 2005-05-05. The Unicode Consortium.
http://www.unicode.org/unicode/reports/tr10/tr10-14.html

[Unicode15] Davis, Mark and Dürst, Martin, *Unicode Standard Annex #15: Unicode Normalization Forms, Version 4.1.0*, 2005-03-25, The Unicode Consortium
http://www.unicode.org/reports/tr15/tr15-25.html

[Unicode18] Davis, Mark, *Unicode Technical Standard #18: Unicode Regular Expressions*, The Unicode Consortium
http://www.unicode.org/reports/tr18/

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

[XQueryFO] (Candidate Recommendation) XQuery 1.0 and XPath 2.0 Functions and Operators, 3 November 2005, World Wide Web Consortium.
http://www.w3.org/TR/2005/WD-xpathfunctions-20051103/

3 Definitions, notations, and conventions

This Clause is modified by Clause 3, "Definitions, notations, and conventions", in ISO/IEC 9075-3.
This Clause is modified by Clause 3, "Definitions, notations, and conventions", in ISO/IEC 9075-4.
This Clause is modified by Clause 3, "Definitions, notations, and conventions", in ISO/IEC 9075-10.
This Clause is modified by Clause 3, "Definitions, notations, and conventions", in ISO/IEC 9075-11.
This Clause is modified by Clause 3, "Definitions, notations, and conventions", in ISO/IEC 9075-13.
This Clause is modified by Clause 3, "Definitions, notations and conventions", in ISO/IEC 9075-14.

3.1 Definitions

This Subclause is modified by Subclause 3.1, "Definitions", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 3.1, "Definitions", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 3.1, "Definitions", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 3.1, "Definitions", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 3.1, "Definitions", in ISO/IEC 9075-14.

3.1.1 Definitions taken from ISO/IEC 10646

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

3.1.1.1 character

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

3.1.1.2 repertoire

3.1.2 Definitions taken from ISO/IEC 14651

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

3.1.2.1 collation

3.1.3 Definitions taken from Unicode

For the purposes of this document, the definitions of the following terms given in The Unicode Standard apply:

3.1.3.1 character encoding form

3.1.3.2 code point

3.1.3.3 code unit

3.1.3.4 control character

3.1.3.5 noncharacter

3.1.3.6 normalization

3.1.3.7 transcoding

3.1.4 Definitions taken from ISO 8601

For the purposes of this document, the definitions of the following terms given in ISO 8601 apply:

3.1.4.1 Coordinated Universal Time (UTC)

3.1.4.2 date (date, calendar in ISO 8601)

3.1.5 Definitions taken from Part 1

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

3.1.6 Definitions provided in Part 2

For the purposes of this document, in addition to those definitions taken from other sources, the following definitions apply:

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

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

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

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

3.1.6.5 comparable (of a pair of values): Capable of being compared, according to the rules of Subclause 8.2, "<comparison predicate>". In most, but not all, cases, the values of a data type can

be compared one with another. For the specification of comparability of individual data types, see Subclause 4.2, "Character strings", through Subclause 4.10, "Collection types".

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.6.24 REF value: A value that references some site.

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

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

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

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

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

3.1.6.30 result set: A sequence of rows specified by a <cursor specification>, brought into existence by opening a cursor and ranged over by that cursor, and having operational properties of sensitivity, scrollability, holdability, and returnability.

3.1.6.31 **result set sequence:** A sequence of returned result sets.

3.1.6.32 **returned result set:** A result set created during execution of an SQL-invoked procedure and not destroyed when that execution terminates. Such a result set can be accessed by using a cursor other than the one that brought it into existence (a received cursor).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

— U+0009, Horizontal Tabulation

— U+000A, Line Feed

— U+000B, Vertical Tabulation

— U+000C, Form Feed

— U+000D, Carriage Return

— U+0085, Next Line

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

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

The only character that is a member of the Unicode General Category class “Zl” is U+2028 (Line Separator). The only character that is a member of the Unicode General Category class “Zp” is U+2029 (Paragraph Separator). The characters that are members of the Unicode General Category class “Zs” are: U+0020 (Space), U+00A0 (No-Break Space), U+1680 (Ogham Space Mark), U+180E (Mongolian Vowel Separator), U+2000 (En Quad), U+2001 (Em Quad), U+2002 (En Space), U+2003 (Em Space), U+2004 (Three-Per-Em Space), U+2005 (Four-Per-Em Space), U+2006 (Six-Per-Em Space), U+2007 (Figure Space), U+2008 (Punctuation Space), U+2009 (Thin Space), U+200A (Hair Space), U+202F (Narrow No-Break Space), U+205F (Space, Medium Mathematical), and U+3000 (Ideographic Space).

3.1.6.46 **with-return cursor:** A cursor that, when opened, creates a result set that is capable of becoming a returned result set. The WITH RETURN option of <declare cursor>, <dynamic declare cursor>, and <allocate cursor statement> specifies a with-return cursor. WITH RETURN may also be specified in the content of an <attributes variable> in a <prepare statement>, to indicate that the prepared statement, when opened as a dynamic cursor, creates a with-return cursor.

3.1.6.47 **XQuery captured substring:** a captured substring, as defined by [XQueryFO].

3.1.6.48 **XQuery option flag:** a valid value of the `$flags` argument of `fn:matches`, as defined by [XQueryFO]

3.1.6.49 **XQuery regular expression:** a regular expression, as defined by [XQueryFO].

3.1.6.50 **XQuery regular expression parenthesized sub-expression:** a regular expression parenthesized sub-expression, as defined by [XQuery FO].

3.1.6.51 **XQuery replacement string:** a valid value of the `$replacement` argument of `fn:replace`, as defined by [XQueryFO].

3.2 Notation

This Subclause is modified by Subclause 3.2, “Notation”, in ISO/IEC 9075-14.

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

3.3 Conventions

This Subclause is modified by Subclause 3.2, “Conventions”, in ISO/IEC 9075-3.
This Subclause is modified by Subclause 3.1, “Conventions”, in ISO/IEC 9075-4.
This Subclause is modified by Subclause 3.2, “Conventions”, in ISO/IEC 9075-10.
This Subclause is modified by Subclause 3.1, “Conventions”, in ISO/IEC 9075-11.
This Subclause is modified by Subclause 3.2, “Conventions”, in ISO/IEC 9075-13.

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

3.3.1 Use of terms

This Subclause is modified by Subclause 3.1.1, "Use of terms", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 3.2.1, "Use of terms", in ISO/IEC 9075-10.

3.3.1.1 Other terms

This Subclause is modified by Subclause 3.1.1.1, "Other terms", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 3.2.1.1, "Other terms", in ISO/IEC 9075-10.

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

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

(Blank page)

4 Concepts

This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-3.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-4.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-9.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-10.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-11.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-13.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-14.

4.1 Data types

This Subclause is modified by Subclause 4.1, "Data types", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.7, "Data types", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.1, "Data types", in ISO/IEC 9075-14.

4.1.1 General introduction to data types

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

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

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

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

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

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

4.1.2 Naming of predefined types

This Subclause is modified by Subclause 4.1.1, "Naming of predefined types", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.1.1, "Naming of predefined types", in ISO/IEC 9075-14.

SQL defines predefined data types named by the following <key word>s:

CHARACTER

CHARACTER VARYING

CHARACTER LARGE OBJECT

BINARY

BINARY VARYING

BINARY LARGE OBJECT

NUMERIC

DECIMAL

SMALLINT

INTEGER

BIGINT

FLOAT

REAL

DOUBLE PRECISION

BOOLEAN

DATE

TIME

TIMESTAMP

INTERVAL

These names are used in the *type designators* that constitute the *type precedence lists* specified in Subclause 9.5, "Type precedence list determination".

For reference purposes:

— The data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are collectively referred to as *character string types* and the values of character string types are known as *character strings*.

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.3 Non-predefined and non-SQL types

This Subclause is modified by Subclause 4.7.1, “Non-predefined and non-SQL types”, in ISO/IEC 9075-13.

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

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

— *DT* is a direct supertype of *ST*.

— *DT* is a direct subtype of *ST*.

— *DT* is compatible with *ST*.

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

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

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

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

— *TY* is *UDT*.

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

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

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

Each host language has its own data types, which are separate and distinct from SQL data types, even though similar names may be used to describe the data types. Mappings of SQL data types to data types in host languages are described in Subclause 11.50, "<SQL-invoked routine>", and Subclause 21.1, "<embedded SQL host program>". Not every SQL data type has a corresponding data type in every host language.

4.1.4 Comparison and ordering

This Subclause is modified by Subclause 4.1.2, "Comparison and ordering", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.1.2, "Comparison and ordering", in ISO/IEC 9075-14.

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

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

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

— *T* is a member of *S*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

— *DT* is a constituent of *DT*.

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

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

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

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

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

4.2 Character strings

This Subclause is modified by Subclause 4.2, "Character strings", in ISO/IEC 9075-10.

4.2.1 Introduction to character strings

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

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

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

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

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

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

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

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

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

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

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

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

4.2.2 Comparison of character strings

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

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

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

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

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

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

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

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

4.2.3 Operations involving character strings

4.2.3.1 Regular expression syntaxes

This International Standard utilizes two syntaxes for regular expressions, a POSIX-based regular expression syntax and the XQuery regular expression syntax.

The POSIX-based regular expression syntax is similar to the syntax specified in [POSIX], but is normatively defined in the General Rules of Subclause 8.6, “<similar predicate>”.

The XQuery regular expression syntax is normatively defined in [XQueryFO], which in turn references [Schema2].

4.2.3.2 Operators that operate on character strings and return character strings

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

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

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

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

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

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

then the result is the null value.

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

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

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

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

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

Figure 1 — Operation of <regular expression substring function>

<regex substring function> is a function, SUBSTRING_REGEX, that searches a string for an XQuery regular expression pattern and returns one occurrence of the matching substring.

<regex transliteration> is a function, TRANSLATE_REGEX, that searches a string for an XQuery regular expression pattern and returns the string with either one or every occurrence of the XQuery regular expression pattern replaced using an XQuery replacement string.

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

from the original string and the string returned by the function is the result of inserting the replacement string into the original string at the starting position.

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

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

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

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

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

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

— The name of the transliteration.

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

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

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

4.2.3.3 Other operators involving character strings

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

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

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

<similar predicate> uses the triadic operator SIMILAR (or the inverse, NOT SIMILAR), operating on three character strings and returning a Boolean. SIMILAR determines whether or not a character string “matches” a given “pattern” (also a character string) using a POSIX-based regular expression. The pattern is in the form of a “regular expression”. In this regular expression, certain characters (<left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>) have a special meaning. The optional third argument specifies the “escape character”, for use when one of the special characters or the “escape character” itself is required in the pattern without its special meaning.

<regex position expression> is a function, POSITION_REGEX, that searches a string for an XQuery regular expression pattern and returns an integer indicating the beginning position, or 1 (one) plus the ending position, of one occurrence of the matched substring.

<regex like predicate> is a predicate, LIKE_REGEX, that performs XQuery regular expression matching.

4.2.3.4 Operations involving large object character strings

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

— <null predicate>.

— <like predicate>.

— <similar predicate>.

— <position expression>.

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

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

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

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

— <general set function>.

— <group by clause>.

— <order by clause>.

— <unique constraint definition>.

— <referential constraint definition>.

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

— UNION, INTERSECT, and EXCEPT.

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

All the operations described within Subclause 4.2.3.2, “Operators that operate on character strings and return character strings”, and Subclause 4.2.3.3, “Other operators involving character strings”, are supported for large object character strings.

4.2.4 Character repertoires

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

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

— The name of the character repertoire.

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

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

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

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

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

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

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

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

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

4.2.5 Character encoding forms

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

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

— The name of the character encoding form.

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

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

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

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

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

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

— UTF32 is the character encoding form specified in ISO/IEC 10646, in which each character is encoded as four octets. It is applicable to the UCS character repertoire.

— UTF16 is the character encoding form specified in ISO/IEC 10646, in which each character is encoded as two or four octets. It is applicable to the UCS character repertoire.

— UTF8 is the character encoding form specified in ISO/IEC 10646, in which each character is encoded as from one to four octets. It is applicable to the UCS character repertoire.

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

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

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

UTF8 UTF16 UTF32

4.2.6 Collations

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

— The name of the collation.

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2.7 Character sets

This Subclause is modified by Subclause 4.2.2, "Character sets", in ISO/IEC 9075-10.

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

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

— The name of the character set.

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

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

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

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

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

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

— ASCII_GRAPHIC is a synonym for GRAPHIC_IRV.

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

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

— ASCII_FULL is a synonym for ISO8BIT.

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

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

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

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

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

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

Table 1 — Overview of character sets

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

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

4.2.8 Universal character sets

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

An SQL-implementation may assume that all UCS strings are normalized in one of Normalization Form C (NFC), Normalization Form D (NFD), Normalization Form KC (NFKC) or Normalization Form KD (NFKD), as specified by [Unicode15]. <normalized predicate> may be used to verify the normalization form to which a particular UCS string conforms. Applications may also use <normalize function> to enforce a particular <normal form>. With the exception of <normalize function> and <normalized predicate>, the result of any operation on an unnormalized UCS string is implementation-defined.

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

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

4.3 Binary strings

4.3.1 Introduction to binary strings

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

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

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

— The length or maximum length in octets of the binary string type.

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

4.3.2 Binary string comparison

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

For binary string values other than binary large object string values, it is implementation-defined whether trailing X’00’s are considered significant when comparing two binary string values that are otherwise equivalent.

4.3.3 Operations involving binary strings

4.3.3.1 Operators that operate on binary strings and return binary strings

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

<binary substring function> is a triadic function identical in syntax and semantics to <character substring function> except that its arguments and the returned value are all binary strings.

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

<binary trim function> is a function identical in syntax and semantics to <trim function> except that its arguments and the returned value are all binary strings.

4.3.3.2 Other operators involving binary strings

<length expression> returns the length of a given binary string, as an exact numeric value, in octets.

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

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

Binary large object strings cannot be used in:

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

— <general set function>.

— <group by clause>.

— <order by clause>.

— <unique constraint definition>.

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

— UNION, INTERSECT, and EXCEPT.

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

4.4 Numbers

4.4.1 Introduction to numbers

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

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

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

— The precision of the numeric type.

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

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

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

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

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

4.4.2 Characteristics of numbers

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4.3 Operations involving numbers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2 — Illustration of WIDTH_BUCKET Semantics

4.5 Boolean types

4.5.1 Introduction to Boolean types

The data type boolean comprises the distinct truth values *True* and *False*. Unless prohibited by a NOT NULL constraint, the boolean data type also supports the truth value *Unknown* as the null value. This specification

does not make a distinction between the null value of the boolean data type and the truth value *Unknown* that is the result of an SQL <predicate>, <search condition>, or <boolean value expression>; they may be used interchangeably to mean exactly the same thing.

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

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

4.5.2 Comparison and assignment of booleans

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

4.5.3 Operations involving booleans

4.5.3.1 Operations on booleans that return booleans

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

4.5.3.2 Other operators involving booleans

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

4.6 Datetimes and intervals

4.6.1 Introduction to datetimes and intervals

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

A datetime data type descriptor contains:

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

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

An interval data type descriptor contains:

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

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

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

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

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

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

NOTE 15 — No standard programming language specified by ISO/IEC 9075 has a data type corresponding to the datetime and interval <data type>s specified by this part of ISO/IEC 9075. Whenever an application written in one of those languages must transfer values of those <data type>s from the SQL-implementation to the host language program, those values must be transformed to values of some <data type> for which the programming language has a corresponding data type. Typically, the <cast specification> is used to transform values of those <data type>s to values of some character string type. Similar solutions are required when transferring such values from a host program into the SQL-implementation.

4.6.2 Datetimes

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

Table 2 — Fields in datetime values

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

Keyword	Meaning
TIMEZONE_HOUR	Hour value of time zone displacement
TIMEZONE_MINUTE	Minute value of time zone displacement

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 3 — Datetime data type conversions

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

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

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

or both of type TIMESTAMP (regardless whether WITH TIME ZONE or WITHOUT TIME ZONE is specified or implicit).

4.6.3 Intervals

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

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

Table 4 — Fields in year-month INTERVAL values

Keyword	Meaning
YEAR	Years
MONTH	Months

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

Table 5 — Fields in day-time INTERVAL values

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

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

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

Table 6 — Valid values for fields in INTERVAL values

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

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

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

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

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

4.6.4 Operations involving datetimes and intervals

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

Table 7 — Valid operators involving datetimes and intervals

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

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

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

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

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

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

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

4.7 User-defined types

This Subclause is modified by Subclause 4.8, "User-defined types", in ISO/IEC 9075-13.

4.7.1 Introduction to user-defined types

This Subclause is modified by Subclause 4.8.1, "Introduction to user-defined types", in ISO/IEC 9075-13.

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

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

— The <method name>.

— The <specific method name>.

— The <SQL parameter declaration list>.

— The <returns data type>.

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

— Whether the method is type-preserving.

— The <language clause>.

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

— Whether STATIC or CONSTRUCTOR is specified.

— Whether the method is deterministic.

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

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

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

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

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

— *M* is an original method of *T1*.

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

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

as *M* and *M3* is an original method or overriding method of a type *T3* such that *T2* is a proper supertype of *T3* and *T3* is a supertype of *T1*.

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

4.7.2 User-defined type descriptor

This Subclause is modified by Subclause 4.8.2, "User-defined type descriptor", in ISO/IEC 9075-13.

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

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

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

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

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

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

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

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

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

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

— The transform descriptor of the user-defined type.

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

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

NOTE 16 — "user-defined representation", "derived representation", and "system-defined representation" of a reference type are defined in Subclause 4.9, "Reference types".

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

- The <method name>.
- The <specific method name>.

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

NOTE 17 — The characteristics of an <overriding method specification> other than the <method name>, <SQL parameter declaration list>, and <returns data type> are the same as the characteristics for the corresponding <original method specification>.

4.7.3 Observers and mutators

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

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

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

4.7.4 Constructors

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

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

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

NOTE 18 — SQL-invoked constructor methods are original methods that cannot be overloaded. An SQL-invoked constructor method and a regular SQL-invoked function may exist such that they have equivalent routine names, the types of the first parameter of the method's augmented parameter list and the function's parameter list are the same, and the types of the corresponding remaining parameters (if any) are identical according to the Syntax Rules of Subclause 9.18, "Data type identity".

4.7.5 Subtypes and supertypes

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

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

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

— T_a and T_b are compatible;

— T_a is a direct subtype of T_b; or

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

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

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

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

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

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

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

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

NOTE 19 — Because a subtype family has exactly one maximal supertype, if two types have a common subtype, they shall also have a minimal common supertype. Thus, for every set of types drawn from the same subtype family, there is some member of that family that is the minimal common supertype of all of the types in that set.

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

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

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

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

4.7.6 User-defined type comparison and assignment

This Subclause is modified by Subclause 4.8.3, "User-defined type comparison and assignment", in ISO/IEC 9075-13.

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

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

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

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

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

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

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

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

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

NOTE 20 — Explicit cast functions or attribute comparisons can be used to make both values of the same subtype family or to perform the comparisons on attributes of the user-defined types.

NOTE 21 — "Subtype" and "subtype family" are defined in Subclause 4.7.5, "Subtypes and supertypes".

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

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

4.7.7 Transforms for user-defined types

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

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

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

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

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

— The group name of the transform group.

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

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

4.8 Row types

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

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

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

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

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

4.9 Reference types

4.9.1 Introduction to reference types

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

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

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

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

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

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

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

— The type designator of *RT*.

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

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

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

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

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

4.9.2 Operations involving references

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

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

4.10 Collection types

4.10.1 Introduction to collection types

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

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

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

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

— The type designator of *CT*.

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

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

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

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

4.10.2 Arrays

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

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

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

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

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

4.10.3 Multisets

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

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

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

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

4.10.4 Collection comparison and assignment

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

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

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

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

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

4.10.5 Operations involving arrays

4.10.5.1 Operators that operate on array values and return array elements

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

4.10.5.2 Operators that operate on array values and return array values

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

4.10.6 Operations involving multisets

4.10.6.1 Operators that operate on multisets and return multiset elements

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

4.10.6.2 Operators that operate on multisets and return multisets

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

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

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

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

4.11 Data conversions

This Subclause is modified by Subclause 4.3, "Data conversions", in ISO/IEC 9075-14.

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

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

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

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

— The name of the source data type.

— The name of the target data type.

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

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

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

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

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

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

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

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

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

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

4.12 Domains

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

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

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

— The name of the domain.

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

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

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

4.13 Columns, fields, and attributes

This Subclause is modified by Subclause 4.9, "Columns, fields, and attributes", in ISO/IEC 9075-9.

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

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

NOTE 22 — The declared type includes the null value and values in proper subtypes of the declared type.

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

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

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

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

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

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

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

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

— The name of the column.

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

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

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

— The nullability characteristic of *C*.

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

— An indication of whether *C* is updatable or not.

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

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

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

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

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

NOTE 23 — Identity columns and the meaning of “start value” are described in Subclause 4.14.7, “Identity columns”.

— An indication of whether *C* is a generated column or not.

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

NOTE 24 — Generated columns and the meaning of “generation expression” are described in Subclause 4.14.8, “Base columns and generated columns”.

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

— The name of the attribute.

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

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

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

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

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

— The name of the field.

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

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

4.14 Tables

This Subclause is modified by Subclause 4.2, "Tables", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.10, "Tables", in ISO/IEC 9075-9.

4.14.1 Introduction to tables

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

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

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

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

4.14.2 Types of tables

This Subclause is modified by Subclause 4.2.1, "Types of tables", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.10.1, "Types of tables", in ISO/IEC 9075-9.

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

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

A derived table is a table derived directly or indirectly from one or more other tables by the evaluation of a <query expression>.

A viewed table is a named derived table defined by a <view definition>. A viewed table is sometimes called a *view*. Base tables and views are identified by <table name>s. The same <table name>, in its fully qualified form, cannot be used for both a base table and a view.

A <query specification>, <table value constructor>, <explicit table>, or <query expression> contained in a <view definition> is called a *view component*.

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

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

A *grouped table* is a set of groups derived during the evaluation of a <group by clause>. A group *G* is a collection of rows in which, for every grouping column *GC*, if the value of *GC* in some row is not distinct from *GV*, then the value of *GC* in every row is *GV*; moreover, if *R1* is a row in group *G1* of grouped table *GT* and *R2* is a row in *GT* such that for every grouping column *GC* the value of *GC* in *R1* is not distinct from the value of *GC* in *R2*, then *R2* is in *G1*. Every row in *GT* is in exactly one group. A group may be considered as a table. Set functions operate on groups.

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

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

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

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

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

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

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

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

4.14.3 Table descriptors

This Subclause is modified by Subclause 4.10.2, “Table descriptors”, in ISO/IEC 9075-9.

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

Every table descriptor includes:

— The column descriptor of each column in the table.

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

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

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

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

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

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

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

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

— The name of the base table.

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

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

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

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

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

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

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

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

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

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

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

— The name of the view.

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

— The original <query expression> of the view.

4.14.4 Relationships between tables

This Subclause is modified by Subclause 4.10.3, "Relationships between tables", in ISO/IEC 9075-9.

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

The *simply underlying table*s of derived tables are defined in the Syntax Rules of Subclause 7.12, "<query specification>", and Subclause 7.13, "<query expression>". A <table or query name> has no simply underlying tables. The simply underlying table of a cursor is the <cursor specification> included in the cursor's result set descriptor (or, for standing cursors, the <cursor specification> included in the cursor declaration descriptor).

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

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

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

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

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

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

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

4.14.5 Referenceable tables, subtables, and supertables

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

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

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

1) T_a and T_b are the same named table.

2) T_a is a direct subtable of T_b.

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

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

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

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

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

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

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

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

4.14.6 Operations involving tables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.14.7 Identity columns

The columns of a base table *BT* can optionally include not more than one *identity column*. The declared type of an identity column is either an exact numeric type with scale 0 (zero), INTEGER for example, or a distinct type whose source type is an exact numeric type with scale 0 (zero). An identity column has a *start value*, an *increment*, a *maximum value*, a *minimum value*, and a *cycle option*. An identity column is associated with an internal sequence generator *SG*. Let *IC* be the identity column of *BT*. When a row *R* is presented for insertion into *BT*, if *R* does not contain a column corresponding to *IC*, then the value *V* for *IC* in the row inserted into *BT* is obtained by applying the General Rules of Subclause 9.23, “Generation of the next value of a sequence generator”, to *SG*. The definition of an identity column may specify GENERATED ALWAYS or GENERATED BY DEFAULT.

NOTE 25 — “Start value”, “increment”, “maximum value”, “minimum value”, and “cycle option” are defined in Subclause 4.21, “Sequence generators”.

NOTE 26 — The notion of an internal sequence generator being associated with an identity column is used only for definitional purposes in this International Standard.

4.14.8 Base columns and generated columns

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

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

4.14.9 Windowed tables

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

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

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

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

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

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

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

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

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

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

— The window name.

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

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

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

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

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

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

— The <window function>s' window structure descriptors have window partitioning clauses that enumerate the same number of column references, and those column references are pairwise equivalent in their order of occurrence; and their window structure descriptors have window ordering clauses with the same number of <sort key>s, and those <sort key>s are all column references, and those column references are pairwise equivalent in their order of occurrence, and the <sort specification>s pairwise specify or imply <collate clause>s that specify equivalent <collation name>s, the same <ordering specification> (ASC or DESC), and the same <null ordering> (NULLS FIRST or NULLS LAST).

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

4.15 Data analysis operations (involving tables)

This Subclause is modified by Subclause 4.4, "Data analysis operations (involving tables)", in ISO/IEC 9075-14.

4.15.1 Introduction to data analysis operations

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

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

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

4.15.2 Group functions

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

A group function is one of:

— The *grouping operation*.

— A *group aggregate function*.

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

4.15.3 Window functions

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

A window function is one of:

— A rank function.

— A distribution function.

— The row number function.

— A window aggregate function.

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

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

NOTE 27 — This implies that if two or more rows are not distinct with respect to the window ordering, then there will be one or more gaps in the sequential rank numbering.

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

NOTE 28 — This implies that there are no gaps in the sequential rank numbering of rows in each window partition.

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

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

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

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

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

4.15.4 Aggregate functions

This Subclause is modified by Subclause 4.4.1, "Aggregate functions", in ISO/IEC 9075-14.

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

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

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

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

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

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

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

— If DISTINCT is specified, then redundant duplicates.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.16 Determinism

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

NOTE 29 — Transaction isolation levels have a significant impact on determinism, particularly isolation levels other than SERIALIZABLE. However, this International Standard does not address that impact, particularly because of the difficulty in clearly specifying that impact without appearing to mandate implementation techniques (such as row or page locking) and because different SQL-implementations almost certainly resolve the issue in significantly different ways.

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

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

4.17 Integrity constraints

4.17.1 Overview of integrity constraints

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

— The name of the constraint.

— An applicable <search condition>.

NOTE 30 — The applicable <search condition> included in the descriptor is not necessarily the <search condition> that might be contained in the SQL-statement whose execution brings the constraint descriptor into existence. The General Rules for the SQL-statement in question specify the applicable <search condition> to be included in the constraint descriptor, in some cases deriving it from a given <search condition>. For example, the syntax for table constraints allows universal quantification over the rows of the table in question to be implicit; in the applicable <search condition> included in the descriptor, that universal quantification is made explicit, to allow for uniform treatment of all types of constraint.

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

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

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

4.17.2 Checking of constraints

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

The checking of a constraint depends on its constraint mode within the current SQL-transaction. Whenever an SQL-statement is executed, every constraint whose mode is immediate is checked, at a certain point after any changes to SQL-data and schemas resulting from that execution have been effected, to see if it is satisfied. A constraint is *satisfied* if and only if the applicable <search condition> included in its descriptor evaluates to *True* or *Unknown*. If any constraint is not satisfied, then any changes to SQL-data or schemas resulting from executing that statement are canceled. (See the General Rules of Subclause 13.5, “<SQL procedure statement>”.

NOTE 31 — This includes SQL-statements that are executed as a direct result or an indirect result of executing a different SQL-statement. It also includes statements whose effects explicitly or implicitly include setting the constraint mode to immediate.

The constraint mode can be set to immediate either explicitly by execution of a <set constraints mode statement>, or implicitly at the end of the current SQL-transaction.

When a <commit statement> is executed, all constraints are effectively checked and, if any constraint is not satisfied, then an exception condition is raised and the SQL-transaction is terminated by an implicit <rollback statement>.

4.17.3 Table constraints

A *table constraint* is a constraint whose descriptor is included in a base table descriptor. Being associated with a particular base table allows for convenient syntactic shorthands in which universal quantification over the rows of the table in question is implied.

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

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

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

In addition to the components of every table constraint descriptor, a unique constraint descriptor includes:

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

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

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

In addition to the components of every table constraint descriptor, a referential constraint descriptor includes:

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

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

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

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

NOTE 32 — If MATCH FULL or MATCH PARTIAL is specified for a referential constraint and if the referencing table has only one column specified in <referential constraint definition> for that referential constraint, or if the referencing table has more than one specified column for that <referential constraint definition>, but none of those columns is nullable, then the effect is the same as if no <match type> were specified.

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

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

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

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

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

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

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

A table check constraint is satisfied if and only if the specified <search condition> evaluates to *True* or *Unknown* for every row of the table to which it applies.

NOTE 33 — Consequently, an empty table satisfies every table check constraint that applies to it.

4.17.4 Domain constraints

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

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

— The template <search condition> for the generation of domain constraint usage <search condition>s.

— A possibly empty set of domain constraint usages.

A domain constraint usage descriptor is created implicitly by the evaluation of a <column definition> whose <data type or domain name> is a <domain name>. If *C* is such a column and *D* is the domain identified by the <domain name>, then every domain constraint *DC* defined for *D* implies a domain constraint usage, to the effect that each value in *C* satisfies *DC*.

In addition to the components of every table constraint descriptor, a domain constraint usage descriptor includes:

— The name of the applicable column.

— The applicable <search condition> that evaluates whether each value in *C* satisfies *DC*.

A domain constraint is satisfied by SQL-data if and only if, for every table *T* that has a column named *C* based on that domain, the applicable <search condition> recorded in the appropriate domain constraint usage evaluates to *True* or *Unknown*.

A domain constraint is satisfied by the result of a <cast specification> if and only if the specified template <search condition>, with each occurrence of the <general value specification> VALUE replaced by that result, evaluates to *True* or *Unknown*.

4.17.5 Assertions

An assertion is a constraint whose descriptor is an independent schema component not included in any table descriptor.

4.18 Functional dependencies

This Subclause is modified by Subclause 4.11, "Functional dependencies", in ISO/IEC 9075-9.

4.18.1 Overview of functional dependency rules and notations

This Subclause is modified by Subclause 4.11.1, "Overview of functional dependency rules and notations", in ISO/IEC 9075-9.

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

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

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

Let "*T*: *A* *B*" (read "in *T*, *A* determines *B*" or "*B* is functionally dependent on *A* in *T*") denote the functional dependency of *B* on *A* in *T*, which is true if, for any possible value of *T*, any two rows that are not distinct for every column in *A* also are not distinct for every column in *B*. When the table *T* is understood from context, the abbreviation "*A* *B*" may also be used.

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

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

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

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

NOTE 34 — Here, "UNION" denotes set union and "DIFFERENCE" denotes set difference.

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

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

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

4.18.2 General rules and definitions

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

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

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

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

NOTE 35 — "BUC" is an acronym for "base table unique constraint", since the starting point of the recursion is a set of known not null columns comprising a nondeferrable unique constraint of a base table.

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

NOTE 36 — A BUC-set may be empty, in which case there is at most one row in the table. This case shall be distinguished from a table with no BUC-set.

An SQL-implementation may define additional rules for determining BUC-sets, provided that every BUC-set *S* of columns of a table *T* shall have an associated base table *BT* such that every column of *S* has a counterpart in *BT*, and for any possible value of the columns of *S*, there is at most one row in *BT* having those values in those columns.

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

NOTE 37 — "BPK" is an acronym for "base table primary key", since the starting point of the recursion is a set of known not null columns comprising a nondeferrable primary key constraint of a base table.

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

NOTE 38 — Like BUC-sets, a BPK-set may be empty.

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

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

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

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

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

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

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

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

4.18.3 Known functional dependencies in a base table

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

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

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

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

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

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

4.18.4 Known functional dependencies in a transition table

Let *TT* be the transition table defined in a trigger *TR* and let *T* be the subject table of *TR*. If *TT* is an old transition table or if *TR* is an AFTER trigger and *TT* is a new transition table, then the BPK-sets, BUC-sets, and known functional dependencies of *TT* are the same as the BPK-sets, BUC-sets, and known functional dependencies of *T*. If *TR* is a BEFORE trigger and *TT* is a new transition table, then no set of columns of *TT* is a BPK-set or a BUC-set and the known functional dependencies of *TT* are the axiomatic functional dependencies.

4.18.5 Known functional dependencies in <table value constructor>

Let *R* be the result of a <table value constructor>, and let *CR* be the set comprising all the columns of *R*.

No set of columns of *R* is a BPK-set or a BUC-set, except as determined by implementation-defined rules.

All axiomatic functional dependencies are *known functional dependencies* of a <table value constructor>. In addition, there may be implementation-defined known functional dependencies (for example, by examining the actual value of the <table value constructor>).

4.18.6 Known functional dependencies in a <joined table>

Let *T1* and *T2* denote the tables identified by the first and second <table reference>s of some <joined table> *JT*. Let *R* denote the table that is the result of *JT*. Let *CT* be the set of columns of the result of *JT*.

Every column of *R* has some counterpart in either *T1* or *T2*. If NATURAL is specified or the <join specification> is a <named columns join>, then some columns of *R* may have counterparts in both *T1* and *T2*.

A set of columns *S* of *R* is a *BPK-set* if *S* has some counterpart in *T1* or *T2* that is a BPK-set, every member of *S* is known not null, and *S* *CT* is a *known functional dependency* of *R*.

A set of columns *S* of *R* is a *BUC-set* if *S* has some counterpart in *T1* or *T2* that is a BUC-set, every member of *S* is known not null, and *S* *CT* is a *known functional dependency* of *R*.

NOTE 39 — The following rules for known functional dependencies in a <joined table> are not mutually exclusive. The set of known functional dependencies is the union of those dependencies generated by all applicable rules, including the rules of deduction presented earlier.

If *A* *B* is a known functional dependency in *T1*, *CA* is the counterpart of *A* in *R*, and *CB* is the counterpart of *B* in *R*, then *CA* *CB* is a known functional dependency in *R* when one of the following is true:

— CROSS, INNER, or LEFT is specified.

— RIGHT or FULL is specified and at least one of the following is true:

- At least one column in *A* is known not nullable.
- All columns in *A* are join partitioning columns.

If *A* *B* is a known functional dependency in *T2*, *CA* is the counterpart of *A* in *R*, and *CB* is the counterpart of *B* in *R*, then *CA* *CB* is a known functional dependency in *R* when one of the following is true:

— CROSS, INNER, or RIGHT is specified.

— LEFT or FULL is specified and at least one of the following is true:

- At least one column in *A* is known not nullable.
- All columns in *A* are join partitioning columns.

If <join condition> is specified, *AP* is an equality AND-component of the <search condition>, one comparand of *AP* is a column reference *CR*, and the other comparand of *AP* is a <literal>, then let *CRC* be the counterparts of *CR* in *R*. Let {} denote the empty set. {} {*CRC*} is a *known functional dependency* in *R* if any of the following conditions is true:

— INNER is specified.

— If LEFT is specified and *CR* is a column reference to a column in *T1*.

— If RIGHT is specified and *CR* is a column reference to a column in *T2*.

NOTE 40 — An SQL-implementation may also choose to recognize {} -> {*CRC*} as a known functional dependency if the other comparand is a deterministic expression containing no column references.

If <join condition> is specified, *AP* is an equality AND-component of the <search condition>, one comparand of *AP* is a column reference *CRA*, and the other comparand of *AP* is a column references *CRB*, then let *CRAC* and *CRBC* be the counterparts of *CRA* and *CRB* in *R*. {*CRAC*} {*CRBC*} is a *known functional dependency* in *R* if any of the following conditions is true:

— INNER is specified.

— If LEFT is specified and *CRA* is a column reference to a column in *T1*.

— If RIGHT is specified and *CRA* is a column reference to a column in *T2*.

NOTE 41 — An SQL-implementation may also choose to recognize the following as known functional dependencies: {*CRAC*} {*CRBC*} if *CRA* is known not nullable, *CRA* is a column of *T1*, and RIGHT or FULL is specified; or if *CRA* is known not nullable, *CRA* is a column of *T2*, and LEFT or FULL is specified.

NOTE 42 — An SQL-implementation may also choose to recognize similar known functional dependencies of the form { CRA_1 , ..., CRA_N } {*CRBC*} in case one comparand is a deterministic expression of column references CRA_1 , ..., CRA_N under similar conditions.

If NATURAL is specified, or if a <join specification> immediately containing a <named columns join> is specified, then let C_1 , ..., C_N be the column names of the corresponding join columns, for i between 1 (one) and *N*. Let *SC* be the <search condition>:

```
( TN1.C1 = TN2.C1 )
AND
...
AND
( TN1.CN = TN2.CN )
```

Let *SLCC* and *SL* be the <select list>s defined in the Syntax Rules of Subclause 7.7, "<joined table>". Let *JT* be the <join type>. Let *TN1* and *TN2* be the exposed <table or query name> or <correlation name> of tables *T1* and *T2*, respectively. Let *IR* be the result of the <query expression>:

```
SELECT SLCC, TN1.*, TN2.*
FROM TN1 JT JOIN TN2
     ON SC
```

The following are recognized as additional *known functional dependencies* of *IR*:

— If INNER or LEFT is specified, then { COALESCE ($TN1.C_i$, $TN2.C_i$) } { $TN1.C_i$ }, for all *i* between 1 (one) and *N*.

— If INNER or RIGHT is specified, then { COALESCE ($TN1.C_i$, $TN2.C_i$) } { $TN2.C_i$ }, for all *i* between 1 (one) and *N*.

The *known functional dependencies* of *R* are the known functional dependencies of:

```
SELECT SL FROM IR
```

4.18.7 Known functional dependencies in a <table primary>

Let *R* be the result of some <table primary> *TP*. The BPK-sets, BUC-sets, and functional dependencies of *R* are determined as follows.

Case:

— If *TP* immediately contains a <table or query name> *TQN* (with or without ONLY), then the counterparts of the BPK-sets and BUC-sets of *TQN* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* *B* is a functional dependency in the result of *TQN*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* *BC* is a *known functional dependency* in *R*.

— If *TP* immediately contains a <derived table> *DT*, then the counterparts of the BPK-sets and BUC-sets of *DT* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* *B* is a functional dependency in the result of *DT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* *BC* is a *known functional dependency* in *R*.

— If *TP* immediately contains a <lateral derived table> *LDT*, then the counterparts of the BPK-sets and BUC-sets of *LDT* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* *B* is a functional dependency in the result of *LDT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* *BC* is a *known functional dependency* in *R*.

— If *TP* immediately contains a <collection derived table> *CDT*, and WITH ORDINALITY is specified, then let *C1* and *C2* be the two columns names of *CDT*. {*C2*} is a BPK-set and a BUC-set, and {*C2*} {*C2*, *C1*} is a *known functional dependency* . If WITH ORDINALITY is not specified, then these rules do not identify any BPK-set, BUC-set, or non-axiomatic known functional dependency.

4.18.8 Known functional dependencies in a <table factor>

Let *R* be the result of <table factor> *TF*. Let *S* be the result of <table primary> immediately contained in *TF*. The counterparts of the BPK-sets and BUC-sets of *S* are the BPK-sets and BUCsets, respectively, of *R*. If *A* *B* is a functional dependency in *S*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* *BC* is a *known functional dependency* in *R*.

4.18.9 Known functional dependencies in a <table reference>

Let *R* be the result of some <table reference> *TR*. The BPK-sets, BUC-sets, and functional dependencies of *R* are determined as follows.

Case:

— If *TR* immediately contains a <table factor> *TF*, then the counterparts of the BPK-sets and BUC-sets of *TF* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* *B* is a functional dependency in the result of *TF*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* *BC* is a *known functional dependency* in *R*.

— If *TR* immediately contains a <joined table> *JT*, then the counterparts of the BPK-sets and BUC-sets of *JT* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* *B* is a functional dependency in the result of *JT*,

and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* *BC* is a *known functional dependency* in *R*.

4.18.10 Known functional dependencies in the result of a <from clause>

Let *R* be the result of some <from clause> *FC*.

If there is only one <table reference> *TR* in *FC*, then the counterparts of the BPK-sets of *TR* and the counterparts of the BUC-sets of *TR* are the BPK-sets and BUC-sets of *TR*, respectively. Otherwise, these rules do not identify any BPK-sets or BUC-sets in the result of *FC*.

If *T* is a <table reference> immediately contained in the <table reference list> of *FC*, then all known functional dependencies in *T* are *known functional dependencies* in *R*.

4.18.11 Known functional dependencies in the result of a <where clause>

Let *T* be the table that is the operand of the <where clause>. Let *R* be the result of the <where clause>. A set of columns *S* in *R* is a *BUC-set* if there is a <table reference> *TR* such that every member of *S* has a counterpart in *TR*, the counterpart of *S* in *TR* is a BUC-set, and *S* *CR*, where *CR* is the set of all columns of *R*. If, in addition, the counterpart of *S* is a BPK-set, then *S* is a *BPK-set*.

If *A* *B* is a known functional dependency in *T*, then let *AC* be the set of columns of *R* whose counterparts are in *A*, and let *BC* be the set of columns of *R* whose counterparts are in *B*. *AC* *BC* is a *known functional dependency* in *R*.

If *AP* is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of *AP* is a column reference *CR*, and the other comparand of *AP* is a <literal>, then let *CRC* be the counterpart of *CR* in *R*. {} {*CRC*} is a *known functional dependency* in R, where {} denotes the empty set.

NOTE 43 — An SQL-implementation may also choose to recognize {} {*CRC*} as a known functional dependency if the other comparand is a deterministic expression containing no column references.

If *AP* is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of *AP* is a column reference *CRA*, and the other comparand of *AP* is a column references *CRB*, then let *CRAC* and *CRBC* be the counterparts of *CRA* and *CRB* in *R*. {*CRBC*} {*CRAC*} and {*CRAC*} {*CRBC*} are *known functional dependencies* in *R*.

NOTE 44 — An SQL-implementation may also choose to recognize known functional dependencies of the form {$CRAC_1$, ..., $CRAC_N$} {*CRBC*} if one comparand is a deterministic expressions that contains column references CRA_1, ..., CRA_N and the other comparand is a column reference *CRB*.

4.18.12 Known functional dependencies in the result of a <group by clause>

Let *T1* be the table that is the operand of the <group by clause>, and let *R* be the result of the <group by clause>.

Let *G* be the set of columns specified by the <grouping column reference list> of the <group by clause>, after applying all syntactic transformations to eliminate ROLLUP, CUBE, and GROUPING SETS.

The columns of *R* are the columns of *G*, with an additional column *CI*, whose value in any particular row of *R* somehow denotes the subset of rows of *T1* that is associated with the combined value of the columns of *G* in that row.

If every element of *G* is a column reference to a known not null column, then *G* is a *BUC-set* of *R*. If *G* is a subset of a BPK-set of columns of *T1*, then *G* is a *BPK-set* of *R*.

G *CI* is a *known functional dependency* in *R*.

NOTE 45 — Any <set function specification> that is specified in conjunction with *R* is necessarily a function of *CI*. If *SFVC* denotes the column containing the results of such a <set function specification>, then *CI* *SFVC* holds true, and it follows that *G* *SFVC* is a *known functional dependency* in the table containing *SFVC*.

4.18.13 Known functional dependencies in the result of a <having clause>

Let *T1* be the table that is the operand of the <having clause>, let *SC* be the <search condition> simply contained in the <having clause>, and let *R* be the result of the <having clause>.

If *S* is a set of columns of *R* and the counterpart of *S* in *T1* is a BPK-set, then *S* is a *BPK-set*. If the counterpart of *S* in *T1* is a BUC-set, then *S* is a *BUC-set*.

Any known functional dependency in the <query expression>

```
SELECT * FROM T1 WHERE SC
```

is a *known functional dependency* in *R*.

4.18.14 Known functional dependencies in a <query specification>

Let *T* be the <table expression> simply contained in the <query specification> *QS* and let *R* be the result of the <query specification>.

Let *SL* be the <select list> of the <query specification>.

Let *T1* be *T* extended to the right with columns arising from <value expression>s contained in the <select list>, as follows: A <value expression> *VE* that is not a column reference specifies a computed column *CC* in *T1*. For every row in *T1*, the value in *CC* is the result of *VE*.

Let *S* be a set of columns of *R* such that every element of *S* arises from the use of <asterisk> in *SL* or by the specification of a column reference as a <value expression> simply contained in *SL*. *S* has counterparts in *T* and *T1*. If the counterpart of *S* in *T* is a BPK-set, then *S* is a *BPK-set*. If the counterpart of *S* in *T* is a BUC-set or a BPK-set, then *S* is a *BUC-set*.

If *A* *B* is some known functional dependency in *T*, then *A* *B* is a *known functional dependency* in *T1*.

Let *CC* be the column specified by some <value expression> *VE* that is not possibly non-deterministic in the <select list>.

Let *OP1*, *OP2*, ... be the operands of *VE* that are column references whose qualifying query is *QS* and that are not contained in an aggregated argument of a <set function specification>.

If *VE* does not contain a <set function specification> whose aggregation query is *QS*, then {*OP1*, *OP2*, ...} *CC* is a *known functional dependency* in *T1*.

If *VE* contains a <set function specification> whose aggregation query is *QS*, then let {*G1*, ...} be the set of grouping columns of *T*. {*G1*, ..., *OP1*, *OP2*, ...} *CC* is a *known functional dependency* in *T1*.

Let *C* *D* be some known functional dependency in *T1*. If all the columns of *C* have counterparts in *R*, then let *DR* be the set comprising those columns of *D* that have counterparts in *R*. *C* *DR* is a *known functional dependency* in *R*.

4.18.15 Known functional dependencies in a <query expression>

If a <with clause> is specified, and RECURSIVE is not specified, then the *BPK-sets*, *BUC-sets*, and *known functional dependencies* of the table identified by a <query name> in the <with list> are the same as the BPK-sets, BUC-sets, and known functional dependencies of the corresponding <query expression>, respectively. If RECURSIVE is specified, then the BPK-sets, BUC-sets, and non-axiomatic known functional dependencies are implementation-defined.

A <query expression> that is a <query term> that is a <query primary> that is a <simple table> is covered by previous Subclauses of this Clause.

If the <query expression> specifies UNION, EXCEPT or INTERSECT, then let *T1* and *T2* be the left and right operand tables and let *R* be the result. Let *CR* be the set comprising all the columns of *R*.

Each column of *R* has a counterpart in *T1* and a counterpart in *T2*.

Case:

— If EXCEPT is specified, then a set *S* of columns of *R* is a *BPK-set* if its counterpart in *T1* is a BPK-set. *S* is a BUC-set if its counterpart in *T1* is a BUC-set.

— If UNION is specified, then there are no BPK-sets and no BUC-sets.

— If INTERSECT is specified, then a set *S* of columns of *R* is a *BPK-set* if either of its counterparts in *T1* and *T2* is a BPK-set. *S* is a *BUC-set* if either of its counterparts in *T1* and *T2* is a BUC-set.

Case:

— If UNION is specified, then no non-axiomatic functional dependency in *T1* or *T2* is a known functional dependency in *R*, apart from any functional dependencies determined by implementation-defined rules.

— If EXCEPT is specified, then all known functional dependencies in *T1* are *known functional dependencies* in *R*.

— If INTERSECT is specified, then all known functional dependencies in *T1* and all known functional dependencies in *T2* are *known functional dependencies* in *R*.

NOTE 46 — Other known functional dependencies may be determined according to implementation-defined rules.

4.19 Candidate keys

If the functional dependency *CK* *CT* holds true in some table *T*, where *CT* consists of all columns of *T*, and there is no proper subset *CK1* of *CK* such that *CK1* *CT* holds true in *T*, then *CK* is a *candidate key* of *T*. The set of candidate keys *SCK* is nonempty because, if no proper subset of *CT* is a candidate key, then *CT* is a candidate key.

NOTE 47 — Because a candidate key is a set (of columns), *SCK* is therefore a set of sets (of columns).

A candidate key *CK* is a *strong candidate key* if *CK* is a BUC-set, or if *T* is a grouped table and *CK* is a subset of the set of grouping columns of *T*. Let *SSCK* be the set of strong candidate keys.

Let *PCK* be the set of *P* such that *P* is a member of *SCK* and *P* is a *BPK-set*.

Case:

— If *PCK* is nonempty, then the *primary key* is chosen from *PCK* as follows: If *PCK* has exactly one element, then that element is the primary key; otherwise, the left-most element of *PCK* is chosen according to the "left-most rule" below. The primary key is also the *preferred candidate key*.

— Otherwise, there is no primary key and the *preferred candidate key* is chosen as follows.

  Case:

  - If *SSCK* has exactly one element, then it is the preferred candidate key; otherwise, if *SSCK* has more than one element, then the left-most element of *SSCK* is chosen, according to the "left-most" rule below.
  - Otherwise, if *SCK* has exactly one element, then it is the preferred candidate key; otherwise, the left-most element of *SCK* is chosen, according to the "left-most" rule below.

— The "left-most" rule:

  - This rule uses the ordering of the columns of a table, as specified elsewhere in this part of ISO/IEC 9075.

    To determine the left-most of two sets of columns of *T*, first list each set in the order of the column-numbers of its members, extending the shorter list with zeros to the length of the longer list. Then, starting at the left of each ordered list, step forward until a pair of unequal column numbers, one from the same position in each list, is found. The list containing the number that is the smaller member of this pair identifies the left-most of the two sets of columns of *T*.

    To determine the left-most of more than two sets of columns of *T*, take the left-most of any two sets, then pair that with one of the remaining sets and take the left-most, and so on until there are no remaining sets.

4.20 SQL-schemas

This Subclause is modified by Subclause 4.3, "SQL-schemas", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.12, "SQL-schemas", in ISO/IEC 9075-9.

An SQL-schema is a persistent descriptor that includes:

— The name of the SQL-schema.

— The <authorization identifier> of the owner of the SQL-schema.

— The name of the default character set for the SQL-schema.

— The <schema path specification> defining the SQL-path for SQL-invoked routines for the SQL-schema.

— The descriptor of every component of the SQL-schema.

In this part of ISO/IEC 9075, the term "schema" is used only in the sense of SQL-schema. The persistent objects described by the descriptors are said to be *owned by* or to have been *created by* the <authorization identifier> of the schema. Each component descriptor is one of:

— A domain descriptor.

— A base table descriptor.

— A view descriptor.

— A constraint descriptor.

— A privilege descriptor.

— A character set descriptor.

— A collation descriptor.

— A transliteration descriptor.

— A user-defined type descriptor.

— A routine descriptor.

— A sequence generator descriptor.

A schema is created initially using a <schema definition> and may be subsequently modified incrementally over time by the execution of <SQL schema statement>s. <schema name>s are unique within a catalog.

A <schema name> is explicitly or implicitly qualified by a <catalog name> that identifies a catalog.

Base tables and views are identified by <table name>s. A <table name> consists of a <schema name>, followed by a <period>, followed by an <identifier>. The <schema name> identifies the schema that includes the table descriptor of the base table or view identified by the <table name>. The <table name>s of base tables and views defined in different schemas can have equivalent <identifier>s.

NOTE 48 — Equivalence of <identifier>s is defined in Subclause 5.2, "<token> and <separator>".

If a reference to a <table name> does not explicitly contain a <schema name>, then a specific <schema name> is implied. The particular <schema name> associated with such a <table name> depends on the context in which the <table name> appears and is governed by the rules for <schema qualified name>.

If a reference to an SQL-invoked routine that is contained in a <routine invocation> does not explicitly contain a <schema name>, then the SQL-invoked routine is selected from the SQL-path of the schema.

The *containing schema* of an <SQL schema statement> is defined as the schema identified by the <schema name> implicitly or explicitly contained in the name of the object that is created or manipulated by that SQL-statement.

4.21 Sequence generators

4.21.1 General description of sequence generators

A *sequence generator* is a mechanism for generating successive exact numeric values, one at a time. A sequence generator is either an *external sequence generator* or an *internal sequence generator*. An external sequence generator is a named schema object while an internal sequence generator is a component of another schema object. A sequence generator has a data type, which shall be an exact numeric type with scale 0 (zero), a minimum value, a maximum value, a start value, an increment, and a cycle option.

Specification of a sequence generator can optionally include the specification of a data type, a minimum value, a maximum value, a start value, an increment, and a cycle option.

If a sequence generator is associated with a negative increment, then it is a *descending sequence generator*; otherwise, it is an *ascending sequence generator*.

A sequence generator has a time-varying *current base value*, which is a value of its data type. A sequence generator has a cycle which consists of all the possible values between the minimum value and the maximum value which are expressible as (current base value + *N* * increment), where *N* is a non-negative number.

When a sequence generator is created, its current base value is initialized to the start value. Subsequently, the current base value is set to the value of the lowest non-issued value in the cycle for an ascending sequence generator, or the highest non-issued value in the cycle for a descending sequence generator.

Any time after a sequence generator is created, its current base value can be set to an arbitrary value of its data type by an <alter sequence generator statement>.

Changes to the current base value of a sequence generator are not controlled by SQL-transactions; therefore, commits and rollbacks of SQL-transactions have no effect on the current base value of a sequence generator.

A sequence generator is described by a sequence generator descriptor. A sequence generator descriptor includes:

— The sequence generator name that is a schema-qualified sequence generator name for an external sequence generator and a zero-length character string for an internal sequence generator.

— The data type descriptor of the data type associated with the sequence generator.

— The increment of the sequence generator.

— The maximum value of the sequence generator.

— The minimum value of the sequence generator.

— The cycle option of the sequence generator.

— The current base value of the sequence generator.

4.21.2 Operations involving sequence generators

When a <next value expression> is applied to a sequence generator *SG*, *SG* issues a value *V* taken from *SG*'s current cycle such that *V* is expressible as the current base value of *SG* plus *N* multiplied by the increment of *SG*, where *N* is a non-negative number.

Thus a sequence generator will normally issue all of the values in its cycle and these will normally be in increasing or decreasing order (depending on the sign of the increment) but within that general ordering separate subgroups of ordered values may occur.

If the sequence generator's cycle is exhausted (*i.e.*, it cannot issue a value that meets the criteria), then a new cycle is created with the current base value set to the minimum value of *SG* (if *SG* is an ascending sequence generator) or the maximum value of *SG* (if *SG* is a descending sequence generator).

If a new cycle is created and the descriptor of *SG* includes NO CYCLE, then an exception condition is raised.

If there are multiple instances of <next value expression>s specifying the same sequence generator within a single SQL-statement, all those instances return the same value for a given row processed by that SQL-statement.

4.22 SQL-client modules

An *SQL-client module* is an SQL-environment object that can include externally-invoked procedures and certain descriptors. An SQL-client module is created and destroyed by implementation-defined mechanisms (which can include the granting and revoking of privileges required for the use of the SQL-client module). An SQL-client module exists in the SQL-environment containing an SQL-client.

If an SQL-client module *S* is defined by an <SQL-client module definition> that contains a <module authorization identifier> *MAI*, then the owner of *S* is *MAI*; otherwise, *S* has no owner.

An SQL-client module can be specified by a <SQL-client module definition> (see Subclause 13.1, “<SQL-client module definition>”).

An SQL-client module includes:

— The name, if any of the SQL-client module.

— The name of the standard programming language from a compilation unit of which an externally-invoked procedure included in the module can be invoked.

— The <module authorization identifier>, if any.

— An indication of whether or not the <module authorization identifier> is to apply to execution of prepared statements resulting from invocation of externally-invoked procedures in the SQL-client module that contain <prepare statement>s or <execute immediate statement>s.

— SQL-client module defaults, for use in the application of Syntax Rules to <externally-invoked procedure>s, <temporary table declaration>s, and <declare cursor>s.

- The name of the schema for use as the default <schema name> when deriving externally-invoked procedures from <externally-invoked procedure>s, specified either by the <schema name> or, failing that, by the <module authorization identifier>.

- The SQL-path, if any, used to qualify:

— Unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in the SQL-client module.

— Unqualified <user-defined type name>s that are immediately contained in <path-resolved user-defined type name>s that are contained in the SQL-client module.

- The names of zero or more *SQL-client module collations*, each specifying a collation for one or more character sets for the SQL-client module.

— The name, if specified, of the character set used to express the <SQL-client module definition>.

NOTE 49 — The <module character set specification> has no effect on the SQL language contained in the SQL-client module and exists only for compatibility with ISO/IEC 9075:1992. It may be used to document the character set of the SQL-client module.

— Module contents:

- Zero or more temporary table descriptors.
- Zero or more cursor declaration descriptors.
- One or more externally-invoked procedures.

A compilation unit is a segment of executable code, possibly consisting of one or more subprograms. An SQL-client module is associated with a compilation unit during its execution. A single SQL-client module may be associated with multiple compilation units and multiple SQL-client modules may be associated with a single compilation unit. The manner in which this association is specified, including the possible requirement for execution of some implementation-defined statement, is implementation-defined. Whether a compilation unit may invoke or transfer control to other compilation units, written in the same or a different programming language, is implementation-defined.

4.23 Embedded syntax

This Subclause is modified by Subclause 4.1, "Embedded syntax", in ISO/IEC 9075-10.

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. This allows database applications to be expressed in a hybrid form in which SQL-statements are embedded directly in a compilation unit. Such a hybrid compilation unit is defined to be equivalent to:

— An SQL-client module, containing externally-invoked procedures and declarations.

— A standard compilation unit in which each SQL-statement has been replaced by an invocation of an externally-invoked procedure in the SQL-client module, and the declarations contained in such SQL-statements have been suitably transformed into declarations in the host language.

If an <embedded SQL host program> contains an <embedded authorization declaration>, then it shall be the first statement or declaration in the <embedded SQL host program>. The <embedded authorization declaration> is not replaced by a procedure or subroutine call of an <externally-invoked procedure>, but is removed and replaced by syntax associated with the <SQL-client module definition>'s <module authorization clause>.

An implementation may reserve a portion of the name space in the <embedded SQL host program> for the names of procedures or subroutines that are generated to replace SQL-statements and for program variables

and branch labels that may be generated as required to support the calling of these procedures or subroutines; whether this reservation is made is implementation-defined. They may similarly reserve name space for the <SQL-client module name> and <procedure name>s of the generated <SQL-client module definition> that may be associated with the resulting standard compilation unit. The portion of the name space to be so reserved, if any, is implementation-defined.

4.24 Dynamic SQL concepts

4.24.1 Overview of dynamic SQL

In many cases, the SQL-statement to be executed can be coded into an <SQL-client module definition> or into a compilation unit using the embedded syntax. In other cases, the SQL-statement is not known when the program is written and will be generated during program execution.

Dynamic execution of SQL-statements can generally be accomplished in two different ways. Statements can be *prepared* for execution and then later executed one or more times; when the statement is no longer needed for execution, it can be *released* by the use of a <deallocate prepared statement>. Alternatively, a statement that is needed only once can be executed without the preparation step—it can be *executed immediately* (not all SQL-statements can be executed immediately).

When a prepared statement is executed, if it has an owner, then it is executed under definer's rights; otherwise, it is executed under invoker's rights.

Many SQL-statements can be written to use “parameters” (which are manifested in static execution of SQL-statements as host parameters in <SQL procedure statement>s contained in <externally-invoked procedure>s in <SQL-client module definition>s or as host variables in <embedded SQL statement>s contained in <embedded SQL host program>s). In SQL-statements that are executed dynamically, the parameters are called dynamic parameters (<dynamic parameter specification>s) and are represented in SQL language by a <question mark> (?).

In many situations, an application that generates an SQL-statement for dynamic execution knows in detail the required characteristics (*e.g.*, <data type>, <length>, <precision>, <scale>, *etc.*) of each of the dynamic parameters used in the statement; similarly, the application may also know in detail the characteristics of the values that will be returned by execution of the statement. However, in other cases, the application may not know this information to the required level of detail; it is possible in some cases for the application to ascertain the information from the Information Schema, but in other cases (*e.g.*, when a returned value is derived from a computation instead of simply from a column in a table, or when dynamic parameters are supplied) this information is not generally available except in the context of preparing the statement for execution.

NOTE 50 — The Information Schema is defined in ISO/IEC 9075-11.

To provide the necessary information to applications, SQL permits an application to request the SQL-server to *describe* a prepared statement. The description of a statement identifies the number of input dynamic parameters (*describe input*) and their data type information or it identifies the number of output dynamic parameters or values to be returned (*describe output*) and their data type information. The description of a statement is placed into the SQL descriptor areas already mentioned.

Many, but not all, SQL-statements can be prepared and executed dynamically.

NOTE 51 — The complete list of statements that may be dynamically prepared and executed is defined in Subclause 4.33.7, "Preparable and immediately executable SQL-statements".

Certain "set statements" (<set catalog statement>, <set schema statement>, <set names statement>, and <set path statement>) have no effect other than to set up default information (catalog name, schema name, character set, and SQL path, respectively) to be applied to other SQL-statements that are prepared or executed immediately or that are invoked directly.

Syntax errors and Access Rule violations caused by the preparation or immediate execution of <preparable statement>s are identified when the statement is prepared (by <prepare statement>) or when it is executed (by <execute statement> or <execute immediate statement>); such violations are indicated by the raising of an exception condition.

4.24.2 Dynamic SQL statements and descriptor areas

An <execute immediate statement> can be used for a one-time preparation and execution of an SQL-statement. A <prepare statement> is used to prepare the generated SQL-statement for subsequent execution. A <deallocate prepared statement> is used to deallocate SQL-statements that have been prepared with a <prepare statement>. A description of the input dynamic parameters for a prepared statement can be obtained by execution of a <describe input statement>. A description of the resultant columns of a <dynamic select statement> or <dynamic single row select statement> can be obtained by execution of a <describe output statement>. A description of the output dynamic parameters of a statement that is neither a <dynamic select statement> nor a <dynamic single row select statement> can be obtained by execution of a <describe output statement>.

For a statement other than a <dynamic select statement>, an <execute statement> is used to associate parameters with the prepared statement and execute it as though it had been coded when the program was written. For a <dynamic select statement>, the prepared <cursor specification> is associated with a declared dynamic cursor via a <dynamic declare cursor> or with an extended statement cursor via an <allocate cursor statement>. The dynamic cursor can be opened and dynamic parameters can be associated with the dynamic cursor with a <dynamic open statement>. Operations on an open dynamic cursor are described in Subclause 4.32.2, "Operations on and using cursors".

The interface for input dynamic parameters and output dynamic parameters for a prepared statement and for the resulting values from a <dynamic fetch statement> or the execution of a prepared <dynamic single row select statement> can be either a list of dynamic parameters or embedded variables or an SQL descriptor area. An SQL descriptor area consists of one or more item descriptor areas, together with a header that includes a count of the number of those item descriptor areas. The header of an SQL descriptor area consists of the fields in Table 23, "Data types of <key word>s used in the header of SQL descriptor areas", in Subclause 20.1, "Description of SQL descriptor areas". Each item descriptor area consists of the fields specified in Table 24, "Data types of <key word>s used in SQL item descriptor areas", in Subclause 20.1, "Description of SQL descriptor areas". The SQL descriptor area is allocated and maintained by the system with the following statements: <allocate descriptor statement>, <deallocate descriptor statement>, <set descriptor statement>, and <get descriptor statement>.

Two kinds of identifier are used for referencing dynamic SQL objects, *extended names* and *non-extended names*. An extended name is an <identifier> assigned to a parameter or variable and the object it identifies is referenced indirectly, by referencing that parameter or variable. A non-extended name is just an <identifier> and the object it identifies is referenced by using that <identifier> directly in an SQL-statement.

SQL descriptor areas are always identified by extended names. Dynamic statements and dynamic cursors can be identified either by non-extended names or by extended names.

Two extended names are equivalent if their values, with leading and trailing <space>s removed, are equivalent according to the rules for <identifier> comparison in Subclause 5.2, "<token> and <separator>".

The *scope* of an extended name is either *global* or *local* and is determined by the run-time context in which the object it identifies is brought into existence.

The scope of a global extended name *GEN* is the SQL-session, meaning that, during the existence of the object *O* it identifies, *GEN* can be used to reference *O* by any SQL-statement executed in that SQL-session.

The scope of a local extended name *LEN* is the SQL-client module *M* containing the externally-invoked procedure that is being executed when the object *O* identified by *LEN* is brought into existence. This means that, during the existence of *O*, *LEN* can be used to reference *O* by any SQL-statement executed in the same SQL-session by an externallyinvoked procedure in *M*.

The scope of a non-extended name is the <SQL-client module definition> containing the SQL-statement that defines it.

NOTE 52 — The namespace of non-extended names is different from the namespace of extended names.

Let *PRP* be the prepared statement resulting from execution of a <prepare statement> in an externally-invoked procedure, SQL-invoked routine, or triggered action *E*. In the following cases, *PRP* has no owner:

— *E* is an SQL-invoked routine whose security characteristic is INVOKED.

— *E* is an externally-invoked procedure contained in an SQL-client module that either has no owner or for which FOR STATIC ONLY was specified.

Otherwise, the owner of *PRP* is the owner of *E*.

4.25 Direct invocation of SQL

Direct invocation of SQL is a mechanism for executing direct SQL-statements, known as <direct SQL statement>s. In direct invocation of SQL, the method of invoking <direct SQL statement>s, the method of raising conditions that result from the execution of <direct SQL statement>s, the method of accessing the diagnostics information that results from the execution of <direct SQL statement>s, and the method of returning the results are implementation-defined.

4.26 Externally-invoked procedures

An externally-invoked procedure consists of an SQL-statement and can be invoked from a compilation unit of a host language. The host language is specified by the <language clause> of the SQL-client module that contains the externally-invoked procedure.

4.27 SQL-invoked routines

This Subclause is modified by Subclause 4.4, "SQL-invoked routines", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.2, "SQL-invoked routines", in ISO/IEC 9075-13.

This Subclause is modified by Subclause 4.5, "SQL-invoked routines", in ISO/IEC 9075-14.

4.27.1 Overview of SQL-invoked routines

This Subclause is modified by Subclause 4.4.1, "Overview of SQL-invoked routines", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.2.1, "Overview of SQL-invoked routines", in ISO/IEC 9075-13.

An *SQL-invoked routine* is an SQL-invoked procedure or an SQL-invoked function. An SQL-invoked routine comprises at least a <schema qualified routine name>, a sequence of <SQL parameter declaration>s, and a <routine body>.

An SQL-invoked routine is an element of an SQL-schema and is called a *schema-level routine*.

An SQL-invoked routine *SR* is said to be *dependent* on a user-defined type *UDT* if *SR* is created during the execution of the <user-defined type definition> that created *UDT* or if *SR* is created during the execution of an <alter type statement> that specifies an <add attribute definition>. An SQL-invoked routine that is dependent on a user-defined type cannot be modified by an <alter routine statement> or be destroyed by a <drop routine statement>. It is destroyed implicitly by a <drop data type statement>.

An *SQL-invoked procedure* is an SQL-invoked routine that is invoked from an SQL <call statement>. An SQL-invoked procedure may have input SQL parameters, output SQL parameters, and SQL parameters that are both input SQL parameters and output SQL parameters. The format of an SQL-invoked procedure is specified by <SQL-invoked procedure> (see Subclause 11.50, "<SQL-invoked routine>").

An SQL-invoked procedure may optionally be specified to require a new savepoint level to be established when it is invoked and destroyed on return from the executed routine body. The alternative of not taking a savepoint can also be directly specified with OLD SAVEPOINT LEVEL. When an SQL-invoked function is invoked a new savepoint level is always established. Savepoint levels are described in Subclause 4.35.2, "Savepoints".

An *SQL-invoked function* is an SQL-invoked routine whose invocation returns a value. Every parameter of an SQL-invoked function is an input SQL parameter, one of which may be designated as the result SQL parameter. The format of an SQL-invoked function is specified by <SQL-invoked function> (see Subclause 11.50, "<SQL-invoked routine>"). An SQL-invoked function can be a *type-preserving function*; a type-preserving function is an SQL-invoked function that has a result SQL parameter. The most specific type of a non-null result of invoking a type-preserving function shall be compatible with the most specific type of the value of the argument substituted for its result SQL parameter.

An *SQL-invoked method* is an SQL-invoked function that is specified by <method specification designator> (see Subclause 11.50, "<SQL-invoked routine>"). There are three kinds of SQL-invoked methods: *SQL-invoked constructor methods*, *instance SQL-invoked methods* and *static SQL-invoked methods*. All SQL-invoked methods are associated with a user-defined type, also known as the *type of the method*. The <method characteristic>s of an SQL-invoked method are specified by a <method specification> contained in the <user-defined type definition> of the type of the method. Both an instance SQL-invoked method and an SQL-invoked constructor method satisfy the following conditions:

— Its first parameter, called the *subject parameter*, has a declared type that is a user-defined type. The type of the subject parameter is the type of the method. A parameter other than the subject parameter is called an *additional parameter*.

— Its descriptor is in the same schema as the descriptor of the data type of its subject parameter.

An SQL-invoked constructor method satisfies the following additional conditions:

— Its <method name> is equivalent to the <qualified identifier> simply contained in the <user-defined type name> included in the user-defined type descriptor of the type of the method.

A static SQL-invoked method satisfies the following conditions:

— It has no subject parameter. Its first parameter, if any, is treated no differently than any other parameter.

— Its descriptor is in the same schema as the descriptor of the structured type of the method. The name of this type (or of some subtype of it) is always specified together with the name of the method when the method is to be invoked.

An SQL-invoked function that is not an SQL-invoked method is an *SQL-invoked regular function*. An SQL-invoked regular function is specified by <function specification> (see Subclause 11.50, "<SQL-invoked routine>").

A *null-call function* is an SQL-invoked function that is defined to return the null value if any of its input arguments is the null value. A null-call function is an SQL-invoked function whose <null-call clause> specifies "RETURNS NULL ON NULL INPUT".

4.27.2 Characteristics of SQL-invoked routines

This Subclause is modified by Subclause 4.2.2, "Characteristics of SQL-invoked routines", in ISO/IEC 9075-13.

An SQL-invoked routine can be an *SQL routine* or an *external routine*. An SQL routine is an SQL-invoked routine whose <language clause> specifies SQL. The <routine body> of an SQL routine is an <SQL procedure statement>; the <SQL procedure statement> forming the <routine body> can be any SQL-statement, including an <SQL control statement>, but excluding an <SQL connection statement> and an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>.

An external routine is one whose <language clause> does not specify SQL. The <routine body> of an external routine is an <external body reference> whose <external routine name> identifies a program written in some standard programming language other than SQL. The program identified by <external routine name> shall not execute either an <SQL connection statement> or an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>

An SQL-invoked routine is uniquely identified by a <specific name>, called the *specific name* of the SQL-invoked routine.

SQL-invoked routines are invoked differently depending on their form. SQL-invoked procedures are invoked by <call statement>s. SQL-invoked regular functions are invoked by <routine invocation>s. Instance SQL-invoked methods are invoked by <method invocation>s, while SQL-invoked constructor methods are invoked by <new specification>s and static SQL-invoked methods are invoked by <static method invocation>s. An invocation of an SQL-invoked routine specifies the <routine name> of the SQL-invoked routine and supplies a sequence of argument values corresponding to the <SQL parameter declaration>s of the SQL-invoked routine. A *subject routine* of an invocation is an SQL-invoked routine that may be invoked by a <routine invocation>. After the selection of the subject routine of a <routine invocation>, the SQL arguments are evaluated and the SQL-invoked routine that will be executed is selected. If the subject routine is an instance SQL-invoked method, then the SQL-invoked routine that is executed is selected from the set of overriding methods of the subject routine. (The term "set of overriding methods" is defined in the General Rules of Subclause 10.4, "<routine invocation>".) The overriding method that is selected is the overriding method with a subject parameter the type designator of whose declared type precedes that of the declared type of the subject parameter of every

other overriding method in the type precedence list of the most specific type of the value of the SQL argument that corresponds to the subject parameter. See the General Rules of Subclause 10.4, "<routine invocation>". If the subject routine is not an SQL-invoked method, then the SQL-invoked routine executed is that subject routine. After the selection of the SQL-invoked routine for execution, the values of the SQL arguments are assigned to the corresponding SQL parameters of the SQL-invoked routine and its <routine body> is executed. If the SQL-invoked routine is an SQL routine, then the <routine body> is an <SQL procedure statement> that is executed according to the General Rules of <SQL procedure statement>. If the SQL-invoked routine is an external routine, then the <routine body> identifies a program written in some standard programming language other than SQL that is executed according to the rules of that standard programming language.

The <routine body> of an SQL-invoked routine is always executed under the same SQL-session from which the SQL-invoked routine was invoked. Before the execution of the <routine body>, a new context for the current SQL-session is created and the values of the current context preserved. When the execution of the <routine body> completes the original context of the current SQL-session is restored.

If the SQL-invoked routine is an external routine, then an effective SQL parameter list is constructed before the execution of the <routine body>. The effective SQL parameter list has different entries depending on the parameter passing style of the SQL-invoked routine. The value of each entry in the effective SQL parameter list is set according to the General Rules of Subclause 10.4, "<routine invocation>", and passed to the program identified by the <routine body> according to the rules of Subclause 13.6, "Data type correspondences". After the execution of that program, if the parameter passing style of the SQL-invoked routine is SQL, then the SQL-implementation obtains the values for output parameters (if any), the value (if any) returned from the program, the value of the SQLSTATE, and the value of the message text (if any) from the values assigned by the program to the effective SQL parameter list. If the parameter passing style of the SQL-invoked routine is GENERAL, then such values are obtained in an implementation-defined manner.

Different SQL-invoked routines can have equivalent <routine name>s. No two SQL-invoked functions in the same schema are allowed to have the same signature. No two SQL-invoked procedures in the same schema are allowed to have the same name and the same number of parameters. Subject routine determination is the process for choosing the subject routine for a given <routine invocation> given a <routine name> and an <SQL argument list>. Subject routine determination for SQL-invoked functions considers the most specific types of all of the arguments (that is, all of the arguments that are not <dynamic parameter specification>s whose types are not known at the time of subject routine determination) to the invocation of the SQL-invoked function in order from left to right. Where there is not an exact match between the most specific types of the arguments and the declared types of the parameters, type precedence lists are used to determine the closest match. See Subclause 9.4, "Subject routine determination".

If a <routine invocation> is contained in a <query expression> of a view, a check constraint, or an assertion, the <triggered action> of a trigger, or in an <SQL-invoked routine>, then the subject routine for that invocation is determined at the time the view is created, the check constraint is defined, the assertion is created, the trigger is created, or the SQL-invoked routine is created. If the subject routine is an SQL-invoked procedure, an SQL-invoked regular function, or a static SQL-invoked method, then the same SQL-invoked routine is executed whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked. If the subject routine is an instance SQL-invoked method, then the SQL-invoked routine that is executed is determined whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked, based on the most specific type of the value resulting from the evaluation of the SQL argument that correspond to the subject parameter. See the General Rules of Subclause 10.4, "<routine invocation>".

All <identifier chain>s in the <routine body> of an SQL routine are resolved to identify the basis and basis referent at the time that the SQL routine is created. Thus, the same columns and SQL parameters are referenced whenever the SQL routine is invoked.

An SQL-invoked routine is either *deterministic* or *possibly non-deterministic*. An SQL-invoked function that is deterministic always returns the identical return value for a given list of SQL argument values. An SQL-invoked procedure that is deterministic always returns the identical values in its output and inout SQL parameters for a given list of SQL argument values. An SQL-invoked routine is possibly non-deterministic if it might produce nonidentical results when invoked with the identical list of SQL argument values.

An external routine *does not possibly contain SQL*, *possibly contains SQL*, *possibly reads SQL-data*, or *possibly modifies SQL-data*. Only an external routine that possibly contains SQL, *possibly reads SQL-data*, or *possibly modifies SQL-data* is permitted to execute SQL-statements during its invocation. Only an SQL-invoked routine that possibly reads SQL-data or possibly modifies SQL-data may read SQL-data during its invocation. Only an SQL-invoked routine that possibly modifies SQL-data may modify SQL-data during its invocation.

An SQL-invoked routine has a *routine authorization identifier*, which is (directly or indirectly) the authorization identifier of the owner of the schema that contains the SQL-invoked routine at the time that the SQL-invoked routine is created.

4.27.3 Execution of SQL-invoked routines

This Subclause is modified by Subclause 4.4.2, "Execution of SQL-invoked routines", in ISO/IEC 9075-4.

When an SQL-invoked routine is invoked, a copy of the current SQL-session context is pushed onto the stack and some values are modified (see the General Rules of Subclause 10.4, "<routine invocation>", before the <routine body> is executed. The treatment of the authorization stack is described in Subclause 4.34.1.1, "SQL-session authorization identifiers".

An SQL-invoked routine has a *routine SQL-path*, which is inherited from its containing SQL-schema, the current SQL-session, or the containing SQL-client module.

An SQL-invoked routine that is an external routine also has an *external routine SQL-path*, which is derived from the <module path specification>, if any, of the <SQL-client module definition> contained in the external program identified by the routine body of the external routine. If that <SQL-client module definition> does not specify a <module path specification>, then the external routine SQL-path is an implementation-defined SQL-path. For both SQL and external routines, the SQL-path of the current SQL-session is used to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>. SQL routines use the routine SQL-path to determine the search order for the subject routines of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>. External routines use the external routine SQL-path to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>.

4.27.4 Routine descriptors

This Subclause is modified by Subclause 4.4.3, "Routine descriptors", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.5.1, "Routine descriptors", in ISO/IEC 9075-14.

An SQL-invoked routine is described by a *routine descriptor*. A routine descriptor includes:

— The routine name of the SQL-invoked routine.

— The specific name of the SQL-invoked routine.

— The routine authorization identifier of the SQL-invoked routine.

— The routine SQL-path of the SQL-invoked routine.

— The name of the language in which the body of the SQL-invoked routine is written.

— For each of the SQL-invoked routine's SQL parameters, the <SQL parameter name>, if it is specified, the <data type>, the ordinal position, and an indication of whether the SQL parameter is an input SQL parameter, an output SQL parameter, or both an input SQL parameter and an output SQL parameter.

— An indication of whether the SQL-invoked routine is an SQL-invoked function or an SQL-invoked procedure.

— If the SQL-invoked routine is an SQL-invoked procedure, then the maximum number of returned result sets.

— An indication of whether the SQL-invoked routine is deterministic or possibly non-deterministic.

— Indications of whether the SQL-invoked routine possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

— If the SQL-invoked routine is an SQL-invoked function, then:

- The <returns data type> of the SQL-invoked function.
- If the <returns data type> simply contains <locator indication>, then an indication that the return value is a locator.
- An indication of whether the SQL-invoked function is a type-preserving function or not.
- An indication of whether the SQL-invoked function is a mutator function or not.
- If the SQL-invoked function is a type-preserving function, then an indication of which parameter is the result parameter.
- An indication of whether the SQL-invoked function is a null-call function.
- An indication of whether the SQL-invoked function is an SQL-invoked method.

— The creation timestamp.

— The last-altered timestamp.

— If the SQL-invoked routine is an SQL routine, then:

- The SQL routine body of the SQL-invoked routine.
- The SQL security characteristic of the SQL routine.

— If the SQL-invoked routine is an external routine, then:

- The external routine name of the external routine.
- The <parameter style> of the external routine.

- If the external routine specifies a <result cast>, then an indication that it specifies a <result cast> and the <data type> specified in the <result cast>. If <result cast> contains <locator indication>, then an indication that the <data type> specified in the <result cast> has a locator indication.
- The external security characteristic of the external routine.
- The external routine SQL-path of the external routine.
- The effective SQL parameter list of the external routine.
- For every SQL parameter that has an associated from-sql function *FSF*, the specific name of *FSF*.
- For every SQL parameter that has an associated to-sql function *TSF*, the specific name of *TSF*.
- If the SQL-invoked routine is an external function and if it has a to-sql function *TRF* associated with the result, then the specific name of *TRF*.
- For every SQL parameter whose <SQL parameter declaration> contains <locator indication>, an indication that the SQL parameter is a locator parameter.

— The schema name of the schema that includes the SQL-invoked routine.

— If the SQL-invoked routine is an SQL-invoked method, then:

- An indication of the user-defined type whose descriptor contains the corresponding method specification descriptor.
- An indication of whether STATIC was specified.

— An indication of whether the SQL-invoked routine is dependent on a user-defined type.

— An indication as to whether or not the SQL-invoked routine requires a new savepoint level to be established when it is invoked.

4.27.5 Result sets returned by SQL-invoked procedures

NOTE 53 — Subclause 3.1.6, "Definitions provided in Part 2", gives definitions of the terms *result set*, *result set sequence*, *returned result set*, and *with-return cursor*, which are used in this Subclause.

An invocation of an SQL-invoked procedure *SIP1* might bring into existence a result set sequence *RSS*. *RSS* consists of the result sets of with-return cursors opened by *SIP1* and remaining open when *SIP1* terminates, placed in the order in which those result sets are created during the execution of *SIP1*.

NOTE 54 — If the same cursor is opened more than once during the execution of *SIP1*, then it is the last opening that is considered to create the result set, even if the result set in question is identical to that created by some earlier opening.

RSS is available through a received cursor, by executing a particular form of <allocate cursor statement>, only to the invoker *INV* of *SIP1*. If *SIP1* is invoked by an SQL-invoked procedure *SIP2*, then *INV* is *SIP2*. If *SIP1* is invoked by an externally-invoked procedure *EIP*, then *INV* is the SQL-client module containing *EIP*. Otherwise, *RSS* is not available.

NOTE 55 — Only the immediate invoker is considered. For example, if an externally-invoked procedure *EIP* executes a <call statement> invoking an SQL-invoked procedure *SIP3* that invokes *SIP1*, then the result set sequence returned by *SIP1* is available only to *SIP3*, until either *SIP3* returns control to *EIP* or another invocation of *SIP1* by *SIP3* is given before *SIP3* returns. There is no mechanism whereby *SIP3* can return *SIP1*'s result set sequence to the invoker of *SIP3*, even if *SIP3* is defined to be able to return a result set sequence.

The invocation of *SIP1* by *INV* destroys any existing result set sequence that might have arisen from some previous invocation of *SIP1* by *INV*. All result set sequences available to *INV* are destroyed when *INV* terminates.

A returned result set includes a sequence of rows called the *constituent rows* and an *initial cursor position*. All rows of a result set sequence are of the same row type, called the *row type* of the result set. The *degree* of a result set is the number of columns in the result set's row type. The number of rows in the result set is its *cardinality*.

The constituent rows and initial cursor position of each returned result set *RS* in *RSS* are determined when *SIP1* returns to *INV*. If the with-return cursor *C* for *RS* is scrollable, then the constituent rows of *RS* are those of the result set of *C* as it exists when *SIP1* terminates; otherwise, the constituent rows are as for scrollable cursors, except that rows preceding the current cursor position of *C* are excluded. If *C* is scrollable, then the initial cursor position of *RS* is the position of *C* when *SIP1* terminates; otherwise, the initial cursor position of *RS* is before the first row.

NOTE 56 — The result set of *C* as it exists when *SIP1* terminates might differ from that generated when *C* was opened, if, for example, any <delete statement: positioned>s or <update statement: positioned>s are executed by *SIP1* before it terminates.

The maximum number of returned result sets that may form a result set sequence is specified by the <returned result sets characteristic> contained in the <SQL-invoked routine> defining *SIP1*. If the actual number of with-return cursors that remain open when *SIP1* returns is greater than the maximum number of returned result sets specified in the <returned result sets characteristic> clause, then a warning condition is raised. It is implementation-dependent whether or not result sets whose positions are greater than that maximum number are returned.

A result set is described by a result set descriptor. A *result set descriptor* includes:

— A <cursor specification>.

— A sequence of rows.

— A position within the sequence of rows (before a row, on a row, or after the last row).

— The operational properties:

- The operational sensitivity property (either SENSITIVE, INSENSITIVE, or ASENSITIVE).
- The operational scrollability property (either SCROLL or NO SCROLL).
- + The operational holdability property (either WITH HOLD or WITHOUT HOLD).
- + The operational returnability property (either WITH RETURN or WITHOUT RETURN).

— If the operational holdability property is WITH HOLD and the position is on a row, then an indication whether a <fetch statement> or <dynamic fetch statement> has been issued since the last <commit statement>.

4.28 SQL-paths

This Subclause is modified by Subclause 4.5, "SQL-paths", in ISO/IEC 9075-4.

An SQL-path is a list of one or more <schema name>s that determines the search order for one of the following:

— The subject routine of a <routine invocation> whose <routine name> does not contain a <schema name>.

— The user-defined type when the <path-resolved user-defined type name> does not contain a <schema name>.

The value specified by CURRENT_PATH is the value of the SQL-path of the current SQL-session. This SQL-path is used to search for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> when the <routine invocation> is contained in <preparable statement>s that are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement>, or contained in <direct SQL statement>s that are invoked directly. The definition of SQL-schemas specifies an SQL-path that is used to search for the subject routine of a <routine invocation> whose <routine name>s do not contain a <schema name> when the <routine invocation> is contained in the <schema definition>.

4.29 Host parameters

This Subclause is modified by Subclause 4.6, "Host parameters", in ISO/IEC 9075-4.

4.29.1 Overview of host parameters

A host parameter is declared in an <externally-invoked procedure> by a <host parameter declaration>. A host parameter either assumes or supplies the value of the corresponding argument in the invocation of the <externally-invoked procedure>.

A <host parameter declaration> specifies the <data type> of its value, which maps to the host language type of its corresponding argument. Host parameters cannot be null, except through the use of indicator parameters.

4.29.2 Status parameters

This Subclause is modified by Subclause 4.6.1, "Status parameters", in ISO/IEC 9075-4.

The SQLSTATE host parameter is a status parameter. It is set to status codes that indicate either that a call of the <externally-invoked procedure> completed successfully or that an exception condition was raised during execution of the <externally-invoked procedure>.

An <externally-invoked procedure> shall specify the SQLSTATE host parameter. The SQLSTATE host parameter is a character string host parameter for which exception values are defined in Clause 24, "Status codes".

If a condition is raised that causes a statement to have no effect other than that associated with raising the condition (that is, not a completion condition), then the condition is said to be an *exception condition* or *exception*. If a condition is raised that permits a statement to have an effect other than that associated with raising the condition (corresponding to an SQLSTATE class value of *successful completion*, *warning*, or *no data*), then the condition is said to be a *completion condition*.

Exception conditions or completion conditions may be raised during the execution of an <SQL procedure statement>. One of the conditions becomes the active condition when the <SQL procedure statement> terminates. If the active condition is an exception condition, then it will be called the active exception condition. If the active condition is a completion condition, then it will be called the active completion condition.

The completion condition *warning* is broadly defined as completion in which the effects are correct, but there is reason to caution the user about those effects. It is raised for implementation-defined conditions as well as

conditions specified in this part of ISO/IEC 9075. The completion condition *no data* has special significance and is used to indicate an empty result. The completion condition *successful completion* is defined to indicate a completion condition that does not correspond to *warning* or *no data*. This includes conditions in which the SQLSTATE subclass provides implementation-defined information of a non-cautionary nature.

For the purpose of choosing status parameter values to be returned, exception conditions for transaction rollback have precedence over exception conditions for statement failure. Similarly, the completion condition *no data* has precedence over the completion condition *warning*, which has precedence over the completion condition *successful completion*. All exception conditions have precedence over all completion conditions. The values assigned to SQLSTATE shall obey these precedence requirements.

4.29.3 Data parameters

A data parameter is a host parameter that is used to either assume or supply the value of data exchanged between a host program and an SQL-implementation.

4.29.4 Indicator parameters

An indicator parameter is an integer host parameter that is specified immediately following another host parameter. Its primary use is to indicate whether the value that the other host parameter assumes or supplies is a null value. An indicator host parameter cannot immediately follow another indicator host parameter.

The other use for indicator parameters is to indicate whether string data truncation occurred during a transfer between a host program and an SQL-implementation in host parameters or host variables. If a non-null string value is transferred and the length of the target is sufficient to accept the entire source value, then the indicator parameter or variable is set to 0 (zero) to indicate that truncation did not occur. However, if the length of the target is insufficient, the indicator parameter or variable is set to the length (in characters or octets, as appropriate) of the source value to indicate that truncation occurred and to indicate original length in characters or octets, as appropriate, of the source.

4.29.5 Locators

A host parameter, a host variable, an SQL parameter of an external routine, or the value returned by an external function may be specified to be a *locator* by specifying AS LOCATOR. A locator is an SQL-session object, rather than SQL-data, that can be used to reference an SQL-data instance. A locator is either a large object locator, a user-defined type locator, an array locator, or a multiset locator.

A large object locator is one of the following:

— Binary large object locator, a value of which identifies a binary large object.

— Character large object locator, a value of which identifies a large object character string.

— National character large object locator, a value of which identifies a national large object character string.

A user-defined type locator identifies a value of the user-defined type specified by the locator specification. An array locator identifies a value of the array type specified by the locator specification. A multiset locator identifies a value of the multiset type specified by the locator specification.

When the value at a site of binary large object type, character large object type, user-defined type, array type, or multiset type is to be assigned to locator of the corresponding type, an implementation-dependent four-octet non-zero integer value is generated and assigned to the target. A locator value uniquely identifies a value of the corresponding type.

A locator may be either *valid* or *invalid*. A host parameter or host variable specified as a locator may be further specified to be a *holdable locator*. When a locator is initially created, it is marked valid and, if applicable, not holdable. A <hold locator statement> identifying the locator shall be specifically executed before the end of the SQL-transaction in which it was created in order to make that locator holdable.

A non-holdable locator remains valid until the end of the SQL-transaction in which it was generated, unless it is explicitly made invalid by the execution of a <free locator statement> or a <rollback statement> that specifies a <savepoint clause> is executed before the end of that SQL-transaction if the locator was generated subsequent to the establishment of the savepoint identified by the <savepoint clause>.

A holdable locator may remain valid beyond the end of the SQL-transaction in which it is generated. A holdable locator becomes invalid whenever a <free locator statement> identifying that locator is executed or the SQL-transaction in which it is generated or any subsequent SQL-transaction is rolled back. All locators remaining valid at the end of an SQL-session are marked invalid when that SQL-session terminates.

4.30 Diagnostics area

This Subclause is modified by Subclause 4.7, "Diagnostics area", in ISO/IEC 9075-4.

A diagnostics area is a place where completion and exception condition information is stored when an SQL-statement is executed. The diagnostics areas associated with an SQL-session form the *diagnostics area stack* of that SQL-session. For definitional purposes, the diagnostics areas in this stack are considered to be numbered sequentially beginning with 1 (one). An additional diagnostics area is maintained by the SQL-client, as described in ISO/IEC 9075-1, Subclause 4.2.3.1, "SQL-clients".

Two operations on diagnostics area stacks are specified in this International Standard for definitional purposes only. *Pushing* a diagnostics area stack effectively creates a new first diagnostics area, incrementing the ordinal position of every existing diagnostics area in the stack by 1 (one). The content of the new first diagnostics area is initially a copy of the content of the old (now second) one. *Popping* a diagnostics area stack effectively destroys the first diagnostics area in the stack and decrements the ordinal position of every remaining diagnostics area by 1 (one). The maximum number of diagnostics areas in a diagnostics area stack is implementation-dependent.

Each diagnostics area consists of a *statement area* and a sequence of one or more *condition areas*, each of which is at any particular time either *occupied* or *vacant*. A diagnostics area is *empty* when each of its condition areas is vacant; *emptying* a diagnostics area brings about this state. A statement area consists of a collection of named *statement information items*. A condition area consists of a collection of named *condition information items*.

A statement information item gives information about the innermost SQL-statement that is being executed when a condition is raised. A condition information item gives information about the condition itself. The names and data types of statement and condition information items are given in Table 30, "<statement information

item name>s for use with <get diagnostics statement>", and in Table 31, "<condition information item name>s for use with <get diagnostics statement>". Their meanings are given by the General Rules of Subclause 23.1, "<get diagnostics statement>".

At the beginning of the execution of any <SQL procedure statement> that is not an <SQL diagnostics statement>, the first diagnostics area is emptied. An implementation places information about a completion condition or an exception condition reported by SQLSTATE into a vacant condition area in this diagnostics area. If other conditions are raised, the extent to which these cause further condition areas to become occupied is implementation-defined.

An <externally-invoked procedure> containing an <SQL diagnostics statement> returns a code indicating a completion or an exception condition for that statement via SQLSTATE, but does not necessarily cause any vacant condition areas to become occupied.

The number of condition areas per diagnostics area is referred to as the *condition area limit*. An SQL-agent may set the condition area limit with the <set transaction statement>; if the SQL-agent does not specify the condition area limit, then the condition area limit is implementation-dependent, but shall be at least one condition area. An SQL-implementation may place information into this area about fewer conditions than there are condition areas. The ordering of the information about conditions placed into a diagnostics area is implementation-dependent, except that the first condition area in a diagnostics area always corresponds to the condition specified by the SQLSTATE value.

The <get diagnostics statement> is used to obtain information from an occupied condition area, referenced by its ordinal position within the first diagnostics area.

4.31 Standard programming languages

This part of ISO/IEC 9075 specifies the actions of <externally-invoked procedure>s in SQL-client modules when those <externally-invoked procedure>s are called by programs that conform to certain specified programming language standards. The term "standard *PLN* program", where *PLN* is the name of a programming language, refers to a program that conforms to the standard for that programming language as specified in Clause 2, "Normative references".

This part of ISO/IEC 9075 specifies a mechanism whereby SQL language may be embedded in programs that otherwise conform to any of the same specified programming language standards.

NOTE 57 — Interfaces between SQL and the Java programming language are defined in ISO/IEC 9075-10 and ISO/IEC 9075-13.

Although there are obviouis mappings between many SQL data types and the data types of most standard programming languages, this is not the case for all SQL data types or for all standard programming languages.

For the purposes of interfacing with programming languages, the data types DATE, TIME, TIMESTAMP, and INTERVAL shall be converted to or from character strings in those programming languages by means of a <cast specification>. It is anticipated that future evolution of programming language standards will support data types corresponding to these four SQL data types; this standard will then be amended to reflect the availability of those corresponding data types.

The data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are also mapped to character strings in the programming languages. However, because the facilities available in the programming languages do not provide the same capabilities as those available in SQL, there shall be agreement between the host program and SQL regarding the specific format of the character data being exchanged. Specific syntax for this agreement is provided in this part of ISO/IEC 9075.

For standard programming languages C and COBOL, BOOLEAN values are mapped to integer variables in the host language. For standard programming languages Ada, Fortran, Pascal, and PL/I, BOOLEAN variables are directly supported.

For the purposes of interfacing with programming languages, the data type ARRAY shall be converted to a locator (see Subclause 4.29.5, "Locators").

For the purposes of interfacing with programming languages, the data type MULTISET shall be converted to a locator (see Subclause 4.29.5, "Locators").

For the purposes of interfacing with programming languages, user-defined types shall be handled with a locator (see Subclause 4.29.5, "Locators") or transformed to another SQL data type that has a defined mapping to the host language (see Subclause 4.7.7, "Transforms for user-defined types").

4.32 Cursors

This Subclause is modified by Subclause 4.9, "Cursors", in ISO/IEC 9075-4.

4.32.1 General description of cursors

This Subclause is modified by Subclause 4.9.1, "General description of cursors", in ISO/IEC 9075-4.

A cursor is a mechanism by which the rows of a table may be acted on (*e.g.*, returned to a host programming language) one at a time.

A cursor is specified by a <declare cursor>, a <dynamic declare cursor>, or an <allocate cursor statement>. A cursor specified by a <declare cursor> is a *standing cursor*. A cursor specified by a <dynamic declare cursor> is a *declared dynamic cursor*. A cursor specified by an <allocate cursor statement> is an *extended dynamic cursor*. An extended dynamic cursor whose <allocate cursor statement> contains a <statement cursor> is an *extended statement cursor*; otherwise, the extended dynamic cursor contains a <result set cursor> and is called a *received cursor*. An extended dynamic cursor is said to be *global* if its extended name is global; otherwise, the extended dynamic cursor is *local*. A *dynamic cursor* is either a declared dynamic cursor or an extended dynamic cursor.

A *declared cursor* is either a standing cursor or a declared dynamic cursor. A declared cursor has a <cursor name>. A <declare cursor> or a <dynamic declare cursor> is immediately contained in the <module contents> of an <SQL-client module definition>. The scope of a <cursor name> is the innermost <SQL-client module definition> *M* that contains it, with the exception of any <SQL schema statement>s contained in *M*.

Every cursor has an associated cursor declaration. For a declared cursor, the cursor declaration is specified by the cursor's <declare cursor> or <dynamic declare cursor>. For an extended dynamic cursor, the cursor declaration is implicitly created by an <allocate cursor statement>.

A cursor declaration is described by a cursor declaration descriptor. A *cursor declaration descriptor* includes:

— The kind of cursor (standing, declared dynamic, extended statement, or received).

— The provenance of the cursor:

- If the cursor is a declared cursor or a local extended dynamic cursor, then an indication of an SQL-client module.
- Otherwise, an SQL-session identifier.

— The name of the cursor:

- If the cursor is a declared cursor, then a <cursor name>.
- Otherwise, an extended name and its scope (GLOBAL or LOCAL).

— The cursor's origin:

- If the cursor is a standing cursor, then the <cursor specification> contained in the <declare cursor>.
- If the cursor is a declared dynamic cursor, then the <statement name> contained in the <dynamic declare cursor>.
- If the cursor is an extended statement cursor, then there is no cursor origin.
- If the cursor is a received cursor, then the cursor's <specific routine designator>.

— The cursor's declared properties:

- The cursor's declared sensitivity property (either SENSITIVE, INSENSITIVE, or ASENSITIVE).
- The cursor's declared scrollability property (either SCROLL or NO SCROLL).
- The cursor's declared holdability property (either WITH HOLD or WITHOUT HOLD).
- The cursor's declared returnability property (either WITH RETURN or WITHOUT RETURN).

A cursor declaration descriptor is identified by the combination of its kind, its provenance, and its name.

A cursor is described by a cursor instance descriptor. A *cursor instance descriptor* includes:

— A cursor declaration descriptor.

— An SQL-session identifier.

— The cursor's state (either open or closed).

— If the cursor is open, then the cursor's result set descriptor.

A cursor instance descriptor is identified by its cursor declaration descriptor and its SQL-session identifier.

The term "cursor", unqualified by either "declaration" or "instance", refers to a cursor instance descriptor.

For every <declare cursor> in an <SQL-client module definition>, a cursor declaration descriptor is included in the SQL-client module, and is effectively created and destroyed at the same time as the SQL-client module. A cursor instance descriptor for the standing cursor is effectively created when an externally-invoked procedure of that externally-invoked procedure of that SQL-client module is invoked. The cursor instance descriptor is destroyed at the end of the SQL-session.

For every <dynamic declare cursor> in an <SQL-client module definition>, a cursor declaration descriptor is included in the SQL-client module, and is effectively created and destroyed at the same time as the SQL-client module. A cursor instance descriptor for the declared dynamic cursor is effectively created when an externally-invoked procedure of that SQL-client module is invoked, and destroyed when a <deallocate prepared statement>

is executed that deallocates the prepared statement on which the declared dynamic cursor is based. The cursor instance descriptor is also destroyed at the end of the SQL-session (if not earlier).

An extended statement cursor is effectively created (*i.e.*, both the cursor declaration descriptor and the cursor instance descriptor) when an <allocate cursor statement> containing a <statement cursor> is executed within an SQL-session and destroyed when that SQL-session is terminated or when a <deallocate prepared statement> is executed that deallocates the prepared statement on which the dynamic cursor is based.

A received cursor is effectively created (*i.e.*, both the cursor declaration descriptor and the cursor instance descriptor) when an <allocate cursor statement> containing a <result set cursor> is executed within an SQL-session and destroyed when that SQL-session is terminated.

A cursor instance descriptor is in either the open state or the closed state. The initial state of a cursor is the closed state. A standing cursor is placed in the open state by an <open statement> and returned to the closed state by a <close statement> or a <rollback statement>. A declared dynamic cursor is placed in the open state by a <dynamic open statement> and returned to the closed state by a <dynamic close statement>. An extended statement cursor is placed in the open state by an <allocate cursor statement> and returned to the closed state by a <dynamic close statement>. A received cursor is placed in the open state by an <allocate cursor statement>, and is advanced to the next result set in the result set sequence by a <dynamic close statement>. After the last result set in the result sequence has been processed, the received cursor is placed in the closed state by a <dynamic close statement>. A received cursor is also placed in the closed state if the associated SQL-invoked procedure is invoked again while the received cursor is open. An open cursor that was not defined as a holdable cursor is also closed by a <commit statement>. (In the case of a received cursor, if the current result set of the received cursor is not holdable, then a <commit statement> causes the received cursor to open on the next result set in the result set sequence, if any.)

A cursor in the open state identifies a result set and a position relative to the ordering of that result set. If the cursor's <cursor specification> does not contain an <order by clause>, or contains an <order by clause> that does not specify the order of the rows completely, then the rows of the result set have an order that is defined only to the extent that the <order by clause> specifies an order and is otherwise implementation-dependent.

NOTE 58 — A definition of “result set” is given in Subclause 3.1.6, “Definitions provided in Part 2”.

When the ordering of a cursor is not defined by an <order by clause>, the relative position of two rows is implementation-dependent. When the ordering of a cursor is partially determined by an <order by clause>, then the relative positions of two rows are determined only by the <order by clause>; if the two rows have equal values for the purpose of evaluating the <order by clause>, then their relative positions are implementation-dependent.

A cursor whose result set’s <cursor specification> contains an <order by clause> is an *ordered cursor*.

A cursor is either *updatable* or *not updatable*. A received cursor is not updatable. For other kinds of cursors, if FOR UPDATE OF is contained in the <cursor specification> *CS* of the result set descriptor of the cursor, or if the <cursor specification> *CS* identified by the cursor is a <query specification> that is simply updatable, FOR READ ONLY and ORDER BY are not contained in *CS*, and the cursor is not scrollable, then the cursor is *updatable*; otherwise, the cursor is *not updatable*. The operations of update and delete are permitted for updatable cursors, subject to constraining Access Rules.

The position of a cursor in the open state is either before a certain row, on a certain row, or after the last row. If a cursor is on a row, then that row is the current row of the cursor. A cursor may be before the first row or after the last row of a result set even though the result set's sequence of rows is empty. When a cursor is initially opened, the position of the cursor is before the first row.

A cursor declaration descriptor and a result set descriptor have four properties: the sensitivity property (either SENSITIVE, INSENSITIVE, or ASENSITIVE), the scrollability property (either SCROLL or NO SCROLL),

the holdability property (either WITH HOLD or WITHOUT HOLD), and the returnability property (either WITH RETURN or WITHOUT RETURN). The four declared cursor properties are set to the explicit or implicit <cursor sensitivity>, <cursor scrollability>, <cursor holdability>, and <cursor returnability>, respectively, of the <cursor properties> contained in the <declare cursor>, <dynamic declare cursor>, or <allocate cursor statement>.

When a cursor is opened, the four declared cursor properties are copied to the result set descriptor's operational properties. In the case of a declared dynamic cursor or an extended statement cursor, the operational properties are modified as specified by the explicit <cursor attributes> of the prepared statement that is the <cursor specification> of the cursor. In the case of a received cursor, the operational sensitivity property is ASENSITIVE, the operational scrollability property is NO SCROLL, the operational holdability property is WITHOUT HOLD, and the operational returnability property is WITHOUT RETURN.

A cursor whose operational sensitivity property is SENSITIVE is said to be *sensitive*; a cursor whose operational sensitivity property is INSENSITIVE is said to be *insensitive*; and a cursor whose operational sensitivity property is ASENSITIVE is said to be *asensitive*.

A cursor whose operational scrollability property is SCROLL is said to be *scrollable*.

A cursor whose operational holdability property is WITH HOLD is said to be *holdable*.

A cursor whose operational returnability property is WITH RETURN is said to be a *with-return* cursor.

A holdable cursor that has been held open retains its position when the new SQL-transaction is initiated. However, if the holdable cursor is a standing cursor, then before either an <update statement: positioned> or a <delete statement: positioned> is permitted to reference that standing cursor in the new SQL-transaction, a <fetch statement> must be issued against the standing cursor (otherwise, an exception condition is raised). If the holdable cursor is a dynamic cursor, then before a <dynamic update statement: positioned>, a <dynamic delete statement: positioned>, a <preparable dynamic update statement: positioned>, or a <preparable dynamic delete statement: positioned> is permitted to reference that dynamic cursor in the new SQL-transaction, a <dynamic fetch statement> must be issued against the dynamic cursor (otherwise an exception condition is raised).

The following aspects of a standing cursor can be determined from the <cursor specification> in its cursor declaration descriptor: whether it is ordered, whether it is updatable, the underlying tables, its sensitivity property, its scrollability property, its holdability property, and its returnability property. Accordingly, terminology regarding these aspects of a standing cursor may also be applied to the cursor declaration descriptor, for example, in the Syntax Rules, Access Rules, and Conformance Rules.

4.32.2 Operations on and using cursors

A <fetch statement> positions an open standing cursor on a specified row of the standing cursor's ordering and retrieves the values of the columns of that row. An <update statement: positioned> updates the current row of the standing cursor. A <delete statement: positioned> deletes the current row of the standing cursor.

A <dynamic fetch statement> positions an open dynamic cursor on a specified row of the dynamic cursor's ordering and retrieves the values of the columns of that row. A <dynamic update statement: positioned> updates the current row of the dynamic cursor. A <dynamic delete statement: positioned> deletes the current row of the dynamic cursor. A <preparable dynamic delete statement: positioned> is used to delete rows through a dynamic cursor when the precise format of the statement is not known until runtime. A <preparable dynamic

update statement: positioned> is used to update rows through a dynamic cursor when the precise format of the statement is not known until runtime.

A *positioned delete statement* is an SQL-statement that is a <delete statement: positioned>, a <dynamic delete statement: positioned>, or a <preparable dynamic delete statement: positioned>.

A *positioned update statement* is an SQL-statement that is a <update statement: positioned>, a <dynamic update statement: positioned>, or a <preparable dynamic update statement: positioned>.

If an error occurs during the execution of an SQL-statement that identifies a cursor, then, except where otherwise explicitly defined, the effect, if any, on the position or state of that cursor is implementation-dependent.

If a completion condition is raised during the execution of an SQL-statement that identifies a cursor, then the particular SQL-statement identifying that open cursor on which the completion condition is returned is implementation-dependent.

The following paragraphs define several terms used to discuss issues relating to a cursor's operational sensitivity property:

A change to SQL-data is said to be *independent* of a cursor *CR* if and only if it is not made by an <update statement: positioned> or a <delete statement: positioned> that is positioned on *CR*.

A change to SQL-data is said to be *significant* to *CR* if and only if it is independent of *CR*, and, had it been committed before *CR* was opened, would have caused the sequence of rows in the result set descriptor of *CR* to be different in any respect.

A change to SQL-data is said to be *visible* to *CR* if and only if it has an effect on the sequence of rows *SR* of the result set descriptor of *CR* by inserting a row in *SR*, deleting a row from *SR*, changing the value of a column of a row of *CR*, or reordering the rows of *SR*.

If a cursor is open, and the SQL-transaction in which the cursor was opened makes a significant change to SQL-data, then whether that change is visible through that cursor before it is closed is determined as follows:

— If the cursor is insensitive, then significant changes are not visible.

— If the cursor is sensitive, then significant changes are visible.

— If the cursor is asensitive, then the visibility of significant changes is implementation-dependent.

If a holdable cursor is open during an SQL-transaction *T* and it is held open for a subsequent SQL-transaction, then whether any significant changes made to SQL-data (by *T* or any subsequent SQL-transaction in which the cursor is held open) are visible through that cursor in the subsequent SQL-transaction before that cursor is closed is determined as follows:

— If the cursor is insensitive, then significant changes are not visible.

— If the cursor is sensitive, then the visibility of significant changes is implementation-defined.

— If the cursor is asensitive, then the visibility of significant changes is implementation-dependent.

If *CR* is a with-return cursor, then the <cursor specification> *CR* included in the result set descriptor of *CR* defines a returned result set. A with-return cursor, if declared in an SQL-invoked procedure and in the open state when the procedure returns to its invoker, yields a returned result set that can be accessed by the invoker of the procedure that generates it.

NOTE 59 — Definitions of “returned result set” and “with-return cursor” are given in Subclause 3.1.6, “Definitions provided in Part 2”.

4.33 SQL-statements

This Subclause is modified by Subclause 4.10, "SQL-statements", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.13, "SQL-statements", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.6, "SQL-statements", in ISO/IEC 9075-14.

4.33.1 Classes of SQL-statements

This Subclause is modified by Subclause 4.10.1, "Classes of SQL-statements", in ISO/IEC 9075-4.

An SQL-statement is a string of characters that conforms to the Format and Syntax Rules specified in the parts of ISO/IEC 9075. Most SQL-statements can be prepared for execution and executed in an SQL-client module, in which case they are prepared when the SQL-client module is created and executed when the containing externally-invoked procedure is called (see Subclause 4.22, "SQL-client modules").

Most SQL-statements can be prepared for execution and executed in additional ways. These are:

— In an embedded SQL host program, in which case they are prepared when the embedded SQL host program is preprocessed (see Subclause 4.23, "Embedded syntax").

— Being prepared and executed by the use of SQL-dynamic statements (which are themselves executed in an SQL-client module or an embedded SQL host program—see Subclause 4.24, "Dynamic SQL concepts").

— Direct invocation, in which case they are effectively prepared immediately prior to execution (see Subclause 4.25, "Direct invocation of SQL").

In this part of ISO/IEC 9075, there are at least six ways of classifying SQL-statements:

— According to their effect on SQL objects, whether persistent objects, *i.e.*, SQL-data, SQL-client modules, and schemas, or transient objects, such as SQL-sessions and other SQL-statements.

— According to whether or not they start an SQL-transaction, or can, or shall, be executed when no SQL-transaction is active.

— According to whether they possibly read SQL-data or possibly modify SQL-data.

— According to whether or not they may be embedded.

— According to whether they may be dynamically prepared and executed.

— According to whether or not they may be directly executed.

This part of ISO/IEC 9075 permits SQL-implementations to provide additional, implementation-defined, statements that may fall into any of these categories. This Subclause will not mention those statements again, as their classification is implementation-defined.

The main classes of SQL-statements are:

— SQL-schema statements; these may have a persistent effect on the set of schemas.

— SQL-data statements; some of these, the SQL-data change statements, may have a persistent effect on SQL-data.

— SQL-transaction statements; except for the <commit statement>, these, and the following classes, have no effects that persist when an SQL-session is terminated.

— SQL-control statements.

— SQL-connection statements.

— SQL-session statements.

— SQL-diagnostics statements.

— SQL-dynamic statements.

— SQL embedded exception declaration.

4.33.2 SQL-statements classified by function

This Subclause is modified by Subclause 4.10.2, "SQL-statements classified by function", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.13.1, "SQL-statements classified by function", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.6.1, "SQL-statements classified by function", in ISO/IEC 9075-14.

4.33.2.1 SQL-schema statements

This Subclause is modified by Subclause 4.10.2.1, "SQL-schema statements", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.13.1.1, "SQL-schema statements", in ISO/IEC 9075-9.

The following are the SQL-schema statements:

— <schema definition>.

— <drop schema statement>.

— <domain definition>.

— <drop domain statement>.

— <table definition>.

— <drop table statement>.

— <view definition>.

— <drop view statement>.

— <assertion definition>.

— <drop assertion statement>.

— <alter table statement>.

— <alter domain statement>.

— <grant privilege statement>.

— <revoke statement>.

— <character set definition>.

— <drop character set statement>.

— <collation definition>.

— <drop collation statement>.

— <transliteration definition>.

— <drop transliteration statement>.

— <trigger definition>.

— <drop trigger statement>.

— <user-defined type definition>.

— <alter type statement>.

— <drop data type statement>.

— <user-defined ordering definition>.

— <drop user-defined ordering statement>.

— <user-defined cast definition>.

— <drop user-defined cast statement>.

— <transform definition>.

— <alter transform statement>.

— <drop transform statement>.

— <schema routine>.

— <alter routine statement>.

— <drop routine statement>.

— <sequence generator definition>.

— <alter sequence generator statement>.

— <drop sequence generator statement>.

— <role definition>.

— <grant role statement>.

— <drop role statement>.

4.33.2.2 SQL-data statements

The following are the SQL-data statements:

— <temporary table declaration>.

— <declare cursor>.

— <open statement>.

— <close statement>.

— <fetch statement>.

— <select statement: single row>.

— <free locator statement>.

— <hold locator statement>.

— <dynamic declare cursor>.

— <allocate cursor statement>.

— <dynamic select statement>.

— <dynamic open statement>.

— <dynamic close statement>.

— <dynamic fetch statement>.

— <direct select statement: multiple rows>.

— <dynamic single row select statement>.

— All SQL-data change statements.

4.33.2.3 SQL-data change statements

The following are the SQL-data change statements:

— <insert statement>.

— <delete statement: searched>.

— <delete statement: positioned>.

— <update statement: searched>.

— <update statement: positioned>.

— <merge statement>.

— <dynamic delete statement: positioned>.

— <preparable dynamic delete statement: positioned>.

— <dynamic update statement: positioned>.

— <preparable dynamic update statement: positioned>.

4.33.2.4 SQL-transaction statements

The following are the SQL-transaction statements:

— <start transaction statement>.

— <set transaction statement>.

— <set constraints mode statement>.

— <commit statement>.

— <rollback statement>.

— <savepoint statement>.

— <release savepoint statement>.

4.33.2.5 SQL-connection statements

The following are the SQL-connection statements:

— <connect statement>.

— <set connection statement>.

— <disconnect statement>.

4.33.2.6 SQL-control statements

This Subclause is modified by Subclause 4.10.2.2, "SQL-control statements", in ISO/IEC 9075-4.

The following are the SQL-control statements:

— <call statement>.

— <return statement>.

4.33.2.7 SQL-session statements

This Subclause is modified by Subclause 4.13.1.2, "SQL-session statements", in ISO/IEC 9075-9.

This Subclause is modified by Subclause 4.6.1.1, "SQL-session statements", in ISO/IEC 9075-14.

The following are the SQL-session statements:

— <set session characteristics statement>.

— <set session user identifier statement>.

— <set role statement>.

— <set local time zone statement>.

— <set catalog statement>.

— <set schema statement>.

— <set names statement>.

— <set path statement>.

— <set transform group statement>.

— <set session collation statement>.

4.33.2.8 SQL-diagnostics statements

This Subclause is modified by Subclause 4.10.2.4, "SQL-diagnostics statements", in ISO/IEC 9075-4.

The following are the SQL-diagnostics statements:

— <get diagnostics statement>.

4.33.2.9 SQL-dynamic statements

The following are the SQL-dynamic statements:

— <execute immediate statement>.

— <allocate descriptor statement>.

— <deallocate descriptor statement>.

— <get descriptor statement>.

— <set descriptor statement>.

— <prepare statement>.

— <deallocate prepared statement>.

— <describe input statement>.

— <describe output statement>.

— <execute statement>.

4.33.2.10 SQL embedded exception declaration

The following is the SQL embedded exception declaration:

— <embedded exception declaration>.

4.33.3 SQL-statements and SQL-data access indication

Some SQL-statements may be classified either as SQL-statements that *possibly read SQL-data* or that *possibly modify SQL-data*. A given SQL-statement belongs to at most one such class.

The following SQL-statements possibly read SQL-data:

— SQL-data statements other than SQL-data change statements, <free locator statement>, and <hold locator statement>.

— SQL-statements that contain a <query expression> and are not SQL-statements that possibly modify SQL-data.

The following SQL-statements possibly modify SQL-data:

— SQL-schema statements.

— SQL-data change statements.

4.33.4 SQL-statements and transaction states

This Subclause is modified by Subclause 4.10.3, "SQL-statements and transaction states", in ISO/IEC 9075-4.

The following SQL-statements are transaction-initiating SQL-statements, *i.e.*, if there is no current SQL-transaction, and an SQL-statement of this class is executed, then an SQL-transaction is initiated, usually before execution of that SQL-statement proceeds:

— All SQL-schema statements

— The following SQL-transaction statements:

- <start transaction statement>.
- <savepoint statement>.
- <commit statement>.
- <rollback statement>.

— The following SQL-data statements:

- <open statement>.
- <close statement>.
- <fetch statement>.
- <select statement: single row>.
- <insert statement>.
- <delete statement: searched>.
- <delete statement: positioned>.
- <update statement: searched>.
- <update statement: positioned>.
- <merge statement>.
- <allocate cursor statement>.
- <dynamic open statement>.
- <dynamic close statement>.
- <dynamic fetch statement>.
- <direct select statement: multiple rows>.
- <dynamic single row select statement>.
- <dynamic delete statement: positioned>.
- <preparable dynamic delete statement: positioned>.
- <dynamic update statement: positioned>.
- <preparable dynamic update statement: positioned>.
- <free locator statement>.
- <hold locator statement>.

— The following SQL-dynamic statements:

- <describe input statement>.
- <describe output statement>.
- <allocate descriptor statement>.
- <deallocate descriptor statement>.
- <get descriptor statement>.
- <set descriptor statement>.
- <prepare statement>.
- <deallocate prepared statement>.

With the exception of <start transaction statement>, every transaction-initiating SQL-statement implicitly initiates an SQL-transaction if there is no current SQL-transaction when it is executed, in which case the execution of the initiating SQL-statement is included in the initiated SQL-transaction. In the case of <start transaction statement>, SQL-transaction initiation is the primary effect of executing the SQL-statement itself.

Whether or not an <execute immediate statement> starts a transaction depends on the content of the <SQL statement variable> referenced by the <execute immediate statement> at the time it is executed. Whether or not an <execute statement> starts a transaction depends on the content of the <SQL statement variable> referenced by the <prepare statement> at the time the prepared statement referenced by the <execute statement> was prepared. In both cases, if the content of the <SQL statement variable> was a transaction-initiating SQL-statement, then the <execute immediate statement> or <execute statement> is treated as a transaction-initiating statement; otherwise, it is not treated as a transaction-initiating statement.

The following SQL-statements are not transaction-initiating SQL-statements, *i.e.*, then executing an SQL-statement of this class does not affect the existence or absence of an SQL-transaction.

— All SQL-transaction statements except <start transaction statement>s, <savepoint statement>s, <commit statement>s, and <rollback statement>s.

— All SQL-connection statements.

— All SQL-session statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <temporary table declaration>.
- <declare cursor>.
- <dynamic declare cursor>.
- <dynamic select statement>.

The following SQL-statements are possibly transaction-initiating SQL-statements:

— <return statement>.

— <call statement>.

If an <SQL control statement> causes the evaluation of a <query expression> and there is no current SQL-transaction, then an SQL-transaction is initiated before evaluation of the <query expression>.

4.33.5 SQL-statement atomicity and statement execution contexts

This Subclause is modified by Subclause 4.10.4, "SQL-statement atomicity and statement execution contexts", in ISO/IEC 9075-4.

The execution of all SQL-statements other than certain SQL-control statements and certain SQL-transaction statements is atomic with respect to recovery. Such an SQL-statement is called an *atomic SQL-statement*. An SQL-statement that is not an atomic SQL-statement is called a *non-atomic SQL statement*.

The following are non-atomic SQL-statements:

— <call statement>

— <execute statement>

— <execute immediate statement>

— <commit statement>

— <return statement>

— <rollback statement>

— <savepoint statement>

All other SQL-statements are atomic SQL-statements.

A statement execution context is either *atomic* or *non-atomic*.

The statement execution context brought into existence by the execution of a non-atomic SQL-statement is a *non-atomic execution context*.

The statement execution context brought into existence by the execution of an atomic SQL-statement or the evaluation of a <subquery> is an *atomic execution context*.

Within one execution context, another execution context may become active. This latter execution context is said to be a *more recent execution context* than the former. If there is no execution context that is more recent than execution context *EC*, then *EC* is said to be the *most recent execution context*.

If there is no atomic execution context that is more recent than atomic execution context *AEC*, then *AEC* is the *most recent atomic execution context*.

An SQL-transaction cannot be explicitly terminated within an atomic execution context. If the execution of an atomic SQL-statement is unsuccessful, then the changes to SQL-data or schemas made by the SQL-statement are canceled.

4.33.6 Embeddable SQL-statements

This Subclause is modified by Subclause 4.10.5, "Embeddable SQL-statements", in ISO/IEC 9075-4.

The following SQL-statements are embeddable in an embedded SQL host program, and may be the <SQL procedure statement> in an <externally-invoked procedure> in an <SQL-client module definition>:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-connection statements.

— All SQL-session statements.

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— The following SQL-data statements:

- <allocate cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <insert statement>.
- <delete statement: searched>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <update statement: searched>.
- <update statement: positioned>.
- <merge statement>.
- <dynamic update statement: positioned>.
- <hold locator statement>.
- <free locator statement>.

— The following SQL-control statements:

- <call statement>.
- <return statement>.

The following SQL-statements are embeddable in an embedded SQL host program, and may occur in an <SQL-client module definition>, though not in an <externally-invoked procedure>:

— <temporary table declaration>.

— <declare cursor>.

— <dynamic declare cursor>.

The following SQL-statements are embeddable in an embedded SQL host program, but may not occur in an <SQL-client module definition>:

— SQL embedded exception declarations.

Consequently, the following SQL-data statements are not embeddable in an embedded SQL host program, nor may they occur in an <SQL-client module definition>, nor be the <SQL procedure statement> in an <externally-invoked procedure> in an <SQL-client module definition>:

— <dynamic select statement>.

— <dynamic single row select statement>.

— <direct select statement: multiple rows>.

— <preparable dynamic delete statement: positioned>.

— <preparable dynamic update statement: positioned>.

4.33.7 Preparable and immediately executable SQL-statements

This Subclause is modified by Subclause 4.10.6, "Preparable and immediately executable SQL-statements", in ISO/IEC 9075-4.

The following SQL-statements are preparable:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-session statements.

— The following SQL-data statements:

- <delete statement: searched>.
- <dynamic select statement>.
- <dynamic single row select statement>.
- <insert statement>.
- <update statement: searched>.
- <merge statement>.
- <preparable dynamic delete statement: positioned>.
- <preparable dynamic update statement: positioned>.
- <preparable implementation-defined statement>.
- <hold locator statement>.
- <free locator statement>.

— The following SQL-control statements:

- <call statement>.

Consequently, the following SQL-statements are not preparable:

— All SQL-connection statements.

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <allocate cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <update statement: positioned>.
- <dynamic update statement: positioned>.
- <direct select statement: multiple rows>.
- <temporary table declaration>.
- <declare cursor>.
- <dynamic declare cursor>.

— The following SQL-control statements:

- <return statement>.

Any preparable SQL-statement can be executed immediately, with the exception of:

— <dynamic select statement>.

— <dynamic single row select statement>.

4.33.8 Directly executable SQL-statements

This Subclause is modified by Subclause 4.10.7, "Directly executable SQL-statements", in ISO/IEC 9075-4.

The following SQL-statements may be executed directly:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-connection statements.

— All SQL-session statements.

— The following SQL-data statements:

- <temporary table declaration>.
- <direct select statement: multiple rows>.
- <insert statement>.
- <delete statement: searched>.
- <update statement: searched>.
- <merge statement>.

— The following SQL-control statements:

- <call statement>.
- <return statement>.

Consequently, the following SQL-statements may not be executed directly:

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <declare cursor>.
- <dynamic declare cursor>.
- <allocate cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <dynamic select statement>.
- <dynamic single row select statement>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <preparable dynamic delete statement: positioned>.

- <update statement: positioned>.
- <dynamic update statement: positioned>.
- <preparable dynamic update statement: positioned>.

— <free locator statement>.

— <hold locator statement>.

4.34 Basic security model

This Subclause is modified by Subclause 4.11, "Basic security model", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.14, "Basic security model", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.7, "Basic security model", in ISO/IEC 9075-14.

4.34.1 Authorization identifiers

An authorization identifier identifies a set of privileges. An authorization identifier is either a user identifier or a role name. A user identifier represents a user of the SQL-implementation. Any mapping of user identifiers to operating system users is implementation-defined. A role name represents a role.

4.34.1.1 SQL-session authorization identifiers

An SQL-session has a user identifier called the *SQL-session user identifier*. When an SQL-session is initiated, the SQL-session user identifier is determined in an implementation-defined manner, unless the session is initiated using a <connect statement>. The value of the SQL-session user identifier can never be the null value. The SQL-session user identifier can be determined by using SESSION_USER.

An SQL-session context contains a time-varying sequence of cells, known as the *authorization stack*, each cell of which contains either a user identifier, a role name, or both. This stack is maintained using a "last-in, first-out" discipline, and effectively only the top cell is visible. When an SQL-session is started, by explicit or implicit execution of a <connect statement>, the authorization stack is initialized with one cell, which contains only the user identifier known as the *SQL-session user identifier*; a role name, known as the *SQL-session role name* may be added subsequently.

Let *E* be an externally-invoked procedure, SQL-invoked routine, triggered action, prepared statement, or directly executed statement. When *E* is invoked, a copy of the top cell is pushed onto the authorization stack. If the invocation of *E* is to be under definer's rights, then the contents of the top cell are replaced with the authorization identifier of the owner of *E*. On completion of the execution of *E*, the top cell is removed.

The contents of the top cell in the authorization stack of the current SQL-session context determine the privileges for the execution of each SQL-statement. The user identifier, if any, in this cell is known as the *current user identifier*; the role name, if any, is known as the *current role name*. They may be determined using CURRENT_USER and CURRENT_ROLE, respectively.

At a given time, there may be no current user identifier or no current role name, but at least one or the other is always present.

NOTE 60 — The privileges granted to PUBLIC are available to all of the <authorization identifier>s in the SQL-environment.

The <set session user identifier statement> changes the value of the current user identifier and of the SQL-session user identifier. The <set role statement> changes the value of the current role name.

The term *current authorization identifier* denotes an authorization identifier in the top cell of the authorization stack.

4.34.1.2 SQL-client module authorization identifiers

If an <SQL-client module definition> contains a <module authorization identifier> *MAI*, then *MAI* is the owner of the corresponding SQL-client module *M* and is used as the current authorization identifier for the execution of each externally-invoked procedure in *M*. If *M* has no owner, then the current user identifier and the current role name of the SQL-session are used as the current user identifier and current role name, respectively, for the execution of each externally-invoked procedure in *M*.

4.34.1.3 SQL-schema authorization identifiers

Every schema has an owner, determined at the time of its creation from a <schema definition> *SD*. That owner is

Case:

— If *SD* simply contains a <schema authorization identifier> *SAI*, then *SAI*.

— If *SD* is simply contained in an <SQL-client module definition> that contains a <module authorization identifier> *MAI*, then *MAI*.

— Otherwise, the SQL-session user identifier.

4.34.2 Privileges

This Subclause is modified by Subclause 4.11.1, "Privileges", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.14.1, "Privileges", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.10, "Privileges", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.7.1, "Privileges", in ISO/IEC 9075-14.

A privilege authorizes a given category of <action> to be performed by a specified <authorization identifier> on a specified object. The object in question is one of the following:

— A base table.

— A view.

— A view component.

— A column.

— A domain.

— A character set.

— A collation.

— A transliteration.

— A user-defined type.

— A table/method pair.

— An SQL-invoked routine.

— A sequence generator.

Each privilege is represented by a *privilege descriptor*. A privilege descriptor contains:

— The identification of the object on which the privilege is granted.

— The <authorization identifier> of the grantor of the privilege.

— The <authorization identifier> of the grantee of the privilege.

— Identification of the <action> that the privilege allows.

— An indication of whether or not the privilege is grantable.

— An indication of whether or not the privilege has the WITH HIERARCHY OPTION specified.

The <action>s that can be specified are:

— INSERT.

— INSERT (<column name list>).

— UPDATE.

— UPDATE (<column name list>).

— DELETE.

— SELECT.

— SELECT (<column name list>).

— SELECT (<privilege method list>).

— REFERENCES.

— REFERENCES (<column name list>).

— USAGE.

— UNDER.

— TRIGGER.

— EXECUTE.

A privilege descriptor with an <action> of INSERT, UPDATE, DELETE, SELECT, TRIGGER, or REFERENCES is called a *table privilege descriptor* and identifies the existence of a privilege on the table or view component identified by the privilege descriptor. If a table privilege descriptor identifies a view component, the privilege descriptor is called a *view component table privilege descriptor*.

A privilege descriptor with an <action> of SELECT (<column name list>), INSERT (<column name list>), UPDATE (<column name list>), or REFERENCES (<column name list>) is called a *column privilege descriptor* and identifies the existence of a privilege on the columns in the table or view component identified by the privilege descriptor. If a column privilege descriptor identifies a view component, the privilege descriptor is called a *view component column privilege descriptor*.

A privilege descriptor with an <action> of SELECT (<privilege method list>) is called a *table/method privilege descriptor* and identifies the existence of a privilege on the methods of the structured type of the table identified by the privilege descriptor.

A table privilege descriptor specifies that the privilege identified by the <action> (unless the <action> is DELETE) is to be automatically granted by the grantor to the grantee on all columns subsequently added to the table.

A privilege descriptor with an <action> of USAGE is called a *usage privilege descriptor* and describes a privilege on an object of one of the following kinds:

— A domain.

— A user-defined type.

— A character set.

— A collation.

— A transliteration.

— A sequence generator.

A privilege descriptor with an <action> of UNDER is called an *under privilege descriptor* and identifies the existence of the privilege on the structured type identified by the privilege descriptor.

A privilege descriptor with an <action> of EXECUTE is called an *execute privilege descriptor* and identifies the existence of a privilege on the SQL-invoked routine identified by the privilege descriptor.

A grantable privilege is a privilege that may be granted by a <grant privilege statement>. The WITH GRANT OPTION clause of a <grant privilege statement> specifies whether the <authorization identifier> identifying the recipient of a privilege (acting as a grantor) may grant it to others.

Privilege descriptors that represent privileges for the owner of an object have a special grantor value, "_SYSTEM". This value is reflected in the Information Schema for all privileges that apply to the owner of the object.

NOTE 61 — The Information Schema is defined in ISO/IEC 9075-11.

A schema that is owned by a given schema <user identifier> or schema <role name> may contain privilege descriptors that describe privileges granted to other <authorization identifier>s (grantees). The granted privileges apply to objects defined in the current schema.

Direct SQL statements are always executed under invoker's rights.

A *view privilege dependency descriptor* is a descriptor that includes two privilege descriptors, called the *supporting privilege descriptor* and the *dependent privilege descriptor*. A view privilege dependency descriptor

is a record that an INSERT, UPDATE, or DELETE privilege of a view, or a column of a view, is directly dependent on another privilege.

4.34.3 Roles

A role, identified by a role name, is, like a user, a potential grantee and grantor of privileges and of other roles. Also like a user, a role can additionally own schemas and other objects.

A role is created by executing a <role definition> and destroyed by executing a <drop role statement>.

A role is granted to one or more authorization identifiers by executing a <grant role statement>, thus conferring on the grantees all the privileges of that role. The granting of a role to an authorization identifier *A* is called a *role authorization* (for *A*).

The privileges of a role with role name *R* are the union of the privileges whose grantee is *R* and the sets of privileges for the role names defined by the role authorizations for *R*. Cycles of role authorizations are prohibited.

The WITH ADMIN OPTION clause of the <grant role statement> for role *R* specifies that each grantee may grant *R* to others, revoke *R* from others, and destroy *R*.

Each role authorization is described by a *role authorization descriptor*. A role authorization descriptor includes:

— The role name of the role.

— The authorization identifier of the grantor.

— The authorization identifier of the grantee.

— An indication of whether or not the role authorization is grantable.

4.34.4 Security model definitions

A role *R* is *applicable for* an authorization identifier *A* if there exists a role authorization descriptor whose role name is *R* and whose grantee is PUBLIC, or *A*, or an applicable role for *A*.

A privilege *P* is *applicable for* an authorization identifier *A* if its grantee is PUBLIC, or *A*, or an applicable role for *A*.

NOTE 62 — “applicable for” is a persistent relationship between persistent objects. Thus, it in no way depends on any SQL-session.

An authorization identifier is *enabled* if it is the current user identifier, the current role name, or a role name that is applicable for the current role name.

A privilege *P* is *current* if *P* is applicable for an enabled authorization identifier.

NOTE 63 — “enabled” and “current” apply to (transient) elements of the current SQL-session context.

4.35 SQL-transactions

This Subclause is modified by Subclause 4.15, "SQL-transactions", in ISO/IEC 9075-9.

4.35.1 General description of SQL-transactions

An *SQL-transaction* (transaction) is a sequence of executions of SQL-statements that is atomic with respect to recovery. These operations are performed by one or more compilation units and SQL-client modules. The operations comprising an SQL-transaction may also be performed by the direct invocation of SQL.

It is implementation-defined whether or not the execution of an SQL-data statement is permitted to occur within the same SQL-transaction as the execution of an SQL-schema statement. If it does occur, then the effect on any open cursor or deferred constraint is implementation-defined. There may be additional implementation-defined restrictions, requirements, and conditions. If any such restrictions, requirements, or conditions are violated, then an implementation-defined exception condition or a completion condition *warning* with an implementation-defined subclass code is raised.

It is implementation-defined whether or not the dynamic execution of an <SQL dynamic data statement> is permitted to occur within the same SQL-transaction as the dynamic execution of an SQL-schema statement. If it does occur, then the effect on any open cursor, prepared dynamic statement, or deferred constraint is implementation-defined. There may be additional implementation-defined restrictions, requirements, and conditions. If any such restrictions, requirements, or conditions are violated, then an implementation-defined exception condition or a completion condition *warning* with an implementation-defined subclass code is raised.

Each SQL-client module that executes an SQL-statement of an SQL-transaction is associated with that SQL-transaction. Each direct invocation of SQL that executes an SQL-statement of an SQL-transaction is associated with that SQL-transaction. An SQL-transaction is initiated when no SQL-transaction is currently active by direct invocation of SQL that results in the execution of a transaction-initiating <direct SQL statement>. An SQL-transaction is initiated when no SQL-transaction is currently active and an <externally-invoked procedure> is called that results in the execution of a *transaction-initiating* SQL-statement. An SQL-transaction is terminated by a <commit statement> or a <rollback statement>. If an SQL-transaction is terminated by successful execution of a <commit statement>, then all changes made to SQL-data or schemas by that SQL-transaction are made persistent and accessible to all concurrent and subsequent SQL-transactions. If an SQL-transaction is terminated by a <rollback statement> or unsuccessful execution of a <commit statement>, then all changes made to SQL-data or schemas by that SQL-transaction are canceled. Committed changes cannot be canceled. If execution of a <commit statement> is attempted, but certain exception conditions are raised, it is unknown whether or not the changes made to SQL-data or schemas by that SQL-transaction are canceled or made persistent.

4.35.2 Savepoints

An SQL-transaction may be partially rolled back by using a savepoint. The savepoint and its <savepoint name> are established within an SQL-transaction when a <savepoint statement> is executed.

An SQL-transaction has one or more *savepoint levels*, exactly one of which is the *current savepoint level*. The savepoint levels of an SQL-transaction are nested, such that when a *new savepoint level NSL is established*, the current savepoint level *CSL* ceases to be current and *NSL* becomes current. When *NSL is destroyed*, *CSL* becomes current again.

A savepoint level exists in an SQL-session *SS* even when no SQL-transaction is active, this savepoint level remaining the current one when an SQL-transaction is initiated in *SS*.

A savepoint *SP* exists at exactly one savepoint level, namely, the savepoint level that is current when *SP* is established.

If a <rollback statement> references a savepoint *SS*, then all changes made to SQL-data or schema subsequent to the establishment of the savepoint are canceled, all savepoints established since *SS* was established are destroyed, and the SQL-transaction is restored to its state as it was immediately following the execution of the <savepoint statement>. Savepoints existing at savepoint level *SPL* are destroyed when *SPL* is destroyed. Savepoint *SS* in the current savepoint level and all savepoints established since *SS* was established are destroyed when a <release savepoint statement> specifying the savepoint name of *SS* is executed. A savepoint may be replaced by another with the same name within a savepoint level by executing a <savepoint statement> that specifies that name.

It is implementation-defined whether or not, or how, a <rollback statement> that references a <savepoint specifier> affects diagnostics area contents, the contents of SQL descriptor areas, and the status of prepared statements.

4.35.3 Properties of SQL-transactions

This Subclause is modified by Subclause 4.15.1, "Properties of SQL-transactions", in ISO/IEC 9075-9.

An SQL-transaction has a *constraint mode* for each integrity constraint. The constraint mode for an integrity constraint in an SQL-transaction is described in Subclause 4.17, "Integrity constraints".

An SQL-transaction has an *access mode* that is either *read-only* or *read-write*. The access mode may be explicitly set by a <set transaction statement> before the start of an SQL-transaction or by the use of a <start transaction statement> to start an SQL-transaction; otherwise, it is implicitly set to the default access mode for the SQL-session before each SQL-transaction begins. If no <set session characteristics statement> has set the default access mode for the SQL-session, then the default access mode for the SQL-session is *read-write*. The term *read-only* applies only to viewed tables and persistent base tables.

An SQL-transaction has a *condition area limit*, which is a positive integer that specifies the maximum number of conditions that can be placed in any diagnostics area during execution of an SQL-statement in this SQL-transaction.

SQL-transactions initiated by different SQL-agents that access the same SQL-data or schemas and overlap in time are *concurrent SQL-transactions*.

4.35.4 Isolation levels of SQL-transactions

An SQL-transaction has an *isolation level* that is READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE. The isolation level of an SQL-transaction defines the degree to which the operations on SQL-data or schemas in that SQL-transaction are affected by the effects of and can affect operations on SQL-data or schemas in concurrent SQL-transactions. The isolation level of an SQL-transaction when any cursor is held open from the previous SQL-transaction within an SQL-session is the isolation level of the previous SQL-transaction by default. If no cursor is held open, or this is the first SQL-transaction within an SQL-session, then the isolation level is SERIALIZABLE by default. The level can be explicitly

set by the <set transaction statement> before the start of an SQL-transaction or by the use of a <start transaction statement> to start an SQL-transaction. If it is not explicitly set, then the isolation level is implicitly set to the default isolation level for the SQL-session before each SQL-transaction begins. If no <set session characteristics statement> has set the default isolation level for the SQL-session, then the default isolation level for the SQL-session is SERIALIZABLE.

Execution of a <set transaction statement> is prohibited after the start of an SQL-transaction and before its termination. Execution of a <set transaction statement> before the start of an SQL-transaction sets the access mode, isolation level, and condition area limit for the single SQL-transaction that is started after the execution of that <set transaction statement>. If multiple <set transaction statement>s are executed before the start of an SQL-transaction, the last such statement is the one whose settings are effective for that SQL-transaction; their actions are not cumulative.

The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.

The isolation level specifies the kind of phenomena that can occur during the execution of concurrent SQL-transactions. The following phenomena are possible:

1) *P1* (“Dirty read”): SQL-transaction *T1* modifies a row. SQL-transaction *T2* then reads that row before *T1* performs a COMMIT. If *T1* then performs a ROLLBACK, *T2* will have read a row that was never committed and that may thus be considered to have never existed.

2) *P2* (“Non-repeatable read”): SQL-transaction *T1* reads a row. SQL-transaction *T2* then modifies or deletes that row and performs a COMMIT. If *T1* then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.

3) *P3* (“Phantom”): SQL-transaction *T1* reads the set of rows *N* that satisfy some <search condition>. SQL-transaction *T2* then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction *T1*. If SQL-transaction *T1* then repeats the initial read with the same <search condition>, it obtains a different collection of rows.

The four isolation levels guarantee that each SQL-transaction will be executed completely or not at all, and that no updates will be lost. The isolation levels are different with respect to phenomena *P1*, *P2*, and *P3*. Table 8, “SQL-transaction isolation levels and the three phenomena” specifies the phenomena that are possible and not possible for a given isolation level.

Table 8 — SQL-transaction isolation levels and the three phenomena

Level	***P1***	***P2***	***P3***
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

NOTE 64 — The exclusion of these phenomena for SQL-transactions executing at isolation level SERIALIZABLE is a consequence of the requirement that such transactions be serializable.

Changes made to SQL-data or schemas by an SQL-transaction in an SQL-session may be perceived by that SQL-transaction in that same SQL-session, and by other SQL-transactions, or by that same SQL-transaction in other SQL-sessions, at isolation level READ UNCOMMITTED, but cannot be perceived by other SQL-transactions at isolation level READ COMMITTED, REPEATABLE READ, or SERIALIZABLE until the former SQL-transaction terminates with a <commit statement>.

Regardless of the isolation level of the SQL-transaction, phenomena *P1*, *P2*, and *P3* shall not occur during the implied reading of schema definitions performed on behalf of executing an SQL-statement, the checking of integrity constraints, and the execution of referential actions associated with referential constraints. The schema definitions that are implicitly read are implementation-dependent. This does not affect the explicit reading of rows from tables in the Information Schema, which is done at the isolation level of the SQL-transaction.

NOTE 65 — The Information Schema is defined in ISO/IEC 9075-11.

4.35.5 Implicit rollbacks

The execution of a <rollback statement> may be initiated implicitly by an SQL-implementation when it detects the inability to guarantee the serializability of two or more concurrent SQL-transactions. When this error occurs, an exception condition is raised: *transaction rollback — serialization failure*.

The execution of a <rollback statement> may be initiated implicitly by an SQL-implementation when it detects unrecoverable errors. When such an error occurs, an exception condition is raised: *transaction rollback* with an implementation-defined subclass code.

The execution of a <rollback statement> may be initiated implicitly when an SQL-implementation detects the loss of the current SQL-Connection. See Subclause 4.36, “SQL-connections”.

4.35.6 Effects of SQL-statements in an SQL-transaction

The execution of an SQL-statement within an SQL-transaction has no effect on SQL-data or schemas other than the effect stated in the General Rules for that SQL-statement, in the General Rules for Subclause 11.8, “<referential constraint definition>”, in the General Rules for Subclause 11.39, “<trigger definition>”, and in the General Rules for Subclause 11.50, “<SQL-invoked routine>”. Together with serializable execution, this implies that all read operations are repeatable within an SQL-transaction at isolation level SERIALIZABLE, except for:

1) The effects of changes to SQL-data or schemas and its contents made explicitly by the SQL-transaction itself.

2) The effects of differences in SQL parameter values supplied to externally-invoked procedures.

3) The effects of references to time-varying system variables such as CURRENT_DATE and CURRENT_USER.

4.35.7 Encompassing transactions

In some environments (*e.g.*, remote database access), an SQL-transaction can be part of an encompassing transaction that is controlled by an agent other than the SQL-agent. The encompassing transaction may involve different resource managers, the SQL-implementation being just one instance of such a manager. In such environments, an encompassing transaction shall be terminated via that other agent, which in turn interacts with the SQL-implementation via an interface that may be different from SQL (COMMIT or ROLLBACK), in order to coordinate the orderly termination of the encompassing transaction. When an SQL-transaction is part of an encompassing transaction that is controlled by an agent other than an SQL-agent and a <rollback statement> is initiated implicitly by an SQL-implementation, then the SQL-implementation will interact with that other agent to terminate that encompassing transaction. The specification of the interface between such agents and the SQL-implementation is beyond the scope of this part of ISO/IEC 9075. However, it is important to note that the semantics of an SQL-transaction remain as defined in the following sense:

— When an agent that is different from the SQL-agent requests the SQL-implementation to rollback an SQL-transaction, the General Rules of Subclause 17.8, "<rollback statement>", are performed.

— When such an agent requests the SQL-implementation to commit an SQL-transaction, the General Rules of Subclause 17.7, "<commit statement>", are performed. To guarantee orderly termination of the encompassing transaction, this commit operation may be processed in several phases not visible to the application; not all the General Rules of Subclause 17.7, "<commit statement>", need to be executed in a single phase.

However, even in such environments, the SQL-agent interacts directly with the SQL-server to set characteristics (such as *read-only* or *read-write*, isolation level, and constraints mode) that are specific to the SQL-transaction model.

It is implementation-defined whether SQL-transactions that affect more than one SQL-server are supported. If such SQL-transactions are supported, then the part of each SQL-transaction that affects a single SQL-server is called a *branch transaction* or a branch of the SQL-transaction. If such SQL-transactions are supported, then they generally have all the same characteristics (access mode, condition area limit, and isolation level, as well as constraint mode). However, it is possible to alter some characteristics of such an SQL-transaction at one SQL-server by the use of the SET LOCAL TRANSACTION statement; if a SET LOCAL TRANSACTION statement is executed at an SQL-server before any transaction-initiating SQL-statement, then it may set the characteristics of that *branch* of the SQL-transaction at that SQL-server.

The characteristics of a branch of an SQL-transaction are limited by the characteristics of the SQL-transaction as a whole:

— If the SQL-transaction is read-write, then the branch of the SQL-transaction may be read-write or read-only; if the SQL-transaction is read-only, then the branch of the SQL-transaction shall be read-only.

— If the SQL-transaction has an isolation level of READ UNCOMMITTED, then the branch of the SQL-transaction may have an isolation level of READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

  If the SQL-transaction has an isolation level of READ COMMITTED, then the branch of the SQL-transaction shall have an isolation level of READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

  If the SQL-transaction has an isolation level of REPEATABLE READ, then the branch of the SQL-transaction shall have an isolation level of REPEATABLE READ or SERIALIZABLE.

  If the SQL-transaction has an isolation level of SERIALIZABLE, then the branch of the SQL-transaction shall have an isolation level of SERIALIZABLE.

— The diagnostics area limit of a branch of an SQL-transaction is always the same as the condition area limit of the SQL-transaction; SET LOCAL TRANSACTION shall not specify a condition area limit.

SQL-transactions that are not part of an encompassing transaction are terminated by the execution of <commit statement>s and <rollback statement>s. If those statements specify AND CHAIN, then they also initiate a new SQL-transaction with the same characteristics as the SQL-transaction that was just terminated, except that the constraint mode of each integrity constraint reverts to its default mode (*deferred* or *immediate*).

4.36 SQL-connections

An *SQL-connection* is an association between an SQL-client and an SQL-server. An SQL-connection may be established and named by a <connect statement>, which identifies the desired SQL-server by means of an <SQL-server name>. A <connection name> is specified as a <simple value specification> whose value is an <identifier>. Two <connection name>s identify the same SQL-connection if their values, with leading and trailing <space>s removed, are equivalent according to the rules for <identifier> comparison in Subclause 5.2, "<token> and <separator>". It is implementation-defined how an SQL-implementation uses <SQL-server name> to determine the location, identity, and communication protocol required to access the SQL-server and create an SQL-session.

An SQL-connection is an *active SQL-connection* if any SQL-statement that initiates or requires an SQL-transaction has been executed at its SQL-server via that SQL-connection during the current SQL-transaction.

An SQL-connection is either *current* or *dormant*. If the SQL-connection established by the most recently executed implicit or explicit <connect statement> or <set connection statement> has not been terminated, then that SQL-connection is the *current SQL-connection*; otherwise, there is no current SQL-connection. An existing SQL-connection that is not the current SQL-connection is a *dormant SQL-connection*.

An SQL implementation may detect the loss of the current SQL-connection during execution of any SQL-statement. When such a connection failure is detected, an exception condition is raised: *transaction rollback — statement completion unknown*. This exception condition indicates that the results of the actions performed in the SQL-server on behalf of the statement are unknown to the SQL-agent.

Similarly, an SQL-implementation may detect the loss of the current SQL-connection during the execution of a <commit statement>. When such a connection failure is detected, an exception condition is raised: *connection exception — transaction resolution unknown*. This exception condition indicates that the SQL-implementation cannot verify whether the SQL-transaction was committed successfully, rolled back, or left active.

A user may initiate an SQL-connection between the SQL-client associated with the SQL-agent and a specific SQL-server by executing a <connect statement>. Otherwise, an SQL-connection between the SQL-client and an implementation-defined default SQL-server is initiated when an externally-invoked procedure is called and no SQL-connection is current. The SQL-connection associated with an implementation-defined default SQL-server is called the *default SQL-connection*. An SQL-connection is terminated either by executing a <disconnect statement>, or following the last call to an externally-invoked procedure within the last active SQL-client module, or by the last execution of a <direct SQL statement> through the direct invocation of SQL. The mechanism and rules by which an SQL-implementation determines whether a call to an externally-invoked procedure is the last call within the last active SQL-client module and the mechanism and rules by which an SQL-implementation determines whether a direct invocation of SQL is the last execution of a <direct SQL statement> are implementation-defined.

An SQL-implementation shall support at least one SQL-connection and may require that the SQL-server be identified at the binding time chosen by the SQL-implementation. If an SQL-implementation permits more

than one concurrent SQL-connection, then the SQL-agent may connect to more than one SQL-server and select the SQL-server by executing a <set connection statement>.

4.37 SQL-sessions

This Subclause is modified by Subclause 4.12, "SQL-sessions", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.16, "SQL-sessions", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.8, "SQL-sessions", in ISO/IEC 9075-14.

4.37.1 General description of SQL-sessions

This Subclause is modified by Subclause 4.12.1, "General description of SQL-sessions", in ISO/IEC 9075-4.

An *SQL-session* spans the execution of a sequence of consecutive SQL-statements invoked either by a single user from a single SQL-agent or by the direct invocation of SQL. At any one time during an SQL-session, exactly one of the SQL-statements in this sequence is being executed and is said to be an *executing statement*. In some cases, an executing statement *ES* causes a nested sequence of consecutive SQL-statements to be executed as a direct result of *ES*; during that time, exactly one of these is also an executing statement and it in turn might similarly involve execution of a further nested sequence, and so on, indefinitely. An executing statement *ES* such that no statement is executing as a direct result of *ES* is called the *innermost executing statement* of the SQL-session.

An SQL-session is associated with an SQL-connection. The SQL-session associated with the default SQL-connection is called the *default SQL-session*. An SQL-session is either *current* or *dormant*. The *current SQL-session* is the SQL-session associated with the current SQL-connection. A *dormant SQL-session* is an SQL-session that is associated with a dormant SQL-connection.

Within an SQL-session, module local temporary tables are effectively created by <temporary table declaration>s. Module local temporary tables are accessible only to invocations of <externally-invoked procedure>s in the SQL-client module in which they are created. The definitions of module local temporary tables persist until the end of the SQL-session.

Within an SQL-session, locators are effectively created when a host parameter, a host variable, or an SQL parameter of an external routine that is specified as a binary large object locator, a character large object locator, a user-defined type locator, an array locator, or a multiset locator is assigned a value of binary large object type, character large object type, user-defined type, array type, or multiset type, respectively. These locators are part of the SQL-session context. A locator may be either valid or invalid. All locators remaining valid at the end of an SQL-session are marked invalid on termination of that SQL-session. A host variable that is a locator may be *holdable* or *nonholdable*.

4.37.2 SQL-session identification

An SQL-session has a unique implementation-dependent SQL-session identifier. This SQL-session identifier is different from the SQL-session identifier of any other concurrent SQL-session. The SQL-session identifier

is used to effectively define implementation-defined schemas that contain the instances of any global temporary tables, created local temporary tables, or declared local temporary tables within the SQL-session.

An SQL-session is started as a result of successful execution of a <connect statement>, which sets the initial SQL-session user identifier to the value of the implicit or explicit <connection user name> contained in the <connect statement>.

An SQL-session initially has no SQL-session role name.

An SQL-session has an original time zone displacement and a current default time zone displacement, which are values of data type INTERVAL HOUR TO MINUTE. Both the original time zone displacement and the current default time zone displacement are initially set to the same implementation-defined value. The current default time zone displacement can subsequently be changed by successful execution of a <set local time zone statement>. The original time zone displacement cannot be changed. It is also possible to set the current default time zone displacement to equal the value of the original time zone displacement.

An SQL-session has a default catalog name that is used to effectively qualify unqualified <schema name>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The default catalog name is initially set to an implementation-defined value but can subsequently be changed by the successful execution of a <set catalog statement> or <set schema statement>. The default catalog name of an SQL-session may be determined by using the <general value specification> CURRENT_CATALOG.

An SQL-session has a default unqualified schema name that is used to effectively qualify unqualified <schema qualified name>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The default unqualified schema name is initially set to an implementation-defined value but can subsequently be changed by the successful execution of a <set schema statement>. The default unqualified schema name of an SQL-session may be determined by using the <general value specification> CURRENT_SCHEMA.

4.37.3 SQL-session properties

This Subclause is modified by Subclause 4.16.1, "SQL-session properties", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.8.1, "SQL-session properties", in ISO/IEC 9075-14.

An SQL-session has an SQL-path that is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The SQL-path is initially set to an implementation-defined value, but can subsequently be changed by the successful execution of a <set path statement>.

The text defining the SQL-path can be referenced by using the <general value specification> CURRENT_PATH.

An SQL-session has a default transform group name and one or more user-defined type name—transform group name pairs that are used to identify the group of transform functions for every user-defined type that is referenced in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The transform group name for a given user-defined type name is initially set

to an implementation-defined value but can subsequently be changed by the successful execution of a <set transform group statement>.

The text defining the transform group names associated with the SQL-session can be referenced using two mechanisms: the <general value specification> “CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>”, which evaluates to the name of the transform group associated with the specified data type, and the <general value specification> “CURRENT_DEFAULT_TRANSFORM_GROUP”, which evaluates to the name of the transform group associated with all types that have no type-specific transform group specified for them.

An SQL-session has a default character set name that is used to identify the character set in which <preparable statement>s are represented when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement>. The default character set name is initially set to an implementation-defined value but can subsequently be changed by the successful execution of a <set names statement>.

For each character set known to the SQL-implementation, an SQL-session has at most one SQL-session collation for that character set, to be used when the rules of Subclause 9.13, “Collation determination”, are applied. There are no SQL-session collations at the start of an SQL-session. The SQL-session collation for a character set can be set or changed by the successful execution of a <set session collation statement>.

An SQL-invoked routine is *active* as soon as an SQL-statement executed by an SQL-agent causes invocation of an SQL-invoked routine and ceases to be active when execution of that invocation is complete.

At any time during an SQL-session, *containing SQL* is said to be *permitted* or *not permitted*. Similarly, *reading SQL-data* is said to be *permitted* or *not permitted* and *modifying SQL-data* is said to be *permitted* or *not permitted*.

An SQL-session has *enduring characteristics*. The enduring characteristics of an SQL-session are initially the same as the default values for the corresponding SQL-session characteristics. The enduring characteristics are changed by successful execution of a <set session characteristics statement> that specifies one or more enduring characteristics. Enduring characteristics that are not specified in a <set session characteristics statement> are not changed in any way by the successful execution of that statement.

An SQL-session has the following enduring characteristics:

— *enduring transaction characteristics*.

Each of the enduring characteristics can be altered at any time in an SQL-session by executing an appropriate <set session characteristics statement>.

An SQL-session has a stack of contexts that is preserved when an SQL-session is made dormant and restored when the SQL-session is made active. Each context in the stack comprises:

— The SQL-session identifier.

— The authorization stack.

— The identities of all instances of temporary tables.

— The original time zone displacement.

— The current default time zone displacement.

— The current constraint mode for each integrity constraint.

— The current access mode.

— The cursor instance descriptor of all open cursors.

— The current isolation level.

— The current SQL diagnostics area stack and its contents, along with the current condition area limit.

— The value of all valid locators.

— The value of the SQL-path for the current SQL-session.

— A *statement execution context*.

— A *routine execution context*.

— Zero or more *trigger execution contexts*.

— All prepared statements prepared during the current SQL-session and not deallocated.

— The current default catalog name.

— The current default unqualified schema name.

— The current default character set name.

— For each character set known to the SQL-implementation, the SQL-session collation, if any.

— The text defining the SQL-path.

— The contents of all SQL dynamic descriptor areas.

— The text defining the default transform group name.

— The text defining the user-defined type name—transform group name pair for each user-defined type explicitly set by the user.

— Each currently available result set sequence *RSS*, along with the specific name of an SQL-invoked procedure *SIP* and the name of the invoker of *SIP* for the invocation causing *RSS* to be brought into existence.

NOTE 66 — Result set sequences are defined in Subclause 4.27.5, "Result sets returned by SQL-invoked procedures".

NOTE 67 — The use of the word "current" in the preceding list implies the values that are current in the SQL-session that is to be made dormant, and not the values that will become current in the SQL-session that will become the active SQL-session.

4.37.4 Execution contexts

Execution contexts augment an SQL-session context to cater for certain special circumstances that might pertain from time to time during invocations of SQL-statements. An execution context is either a statement execution context, a trigger execution context, or a routine execution context. There is always a *statement execution context*, a *routine execution context*, and zero or more *trigger execution contexts*. For certain SQL-statements, the statement execution context is always *atomic*; for others, it is always or sometimes non-atomic. A routine execution context is either atomic or non-atomic. Every trigger execution context is atomic. Statement execution contexts are described in Subclause 4.33.5, "SQL-statement atomicity and statement execution contexts", routine execution contexts in Subclause 4.37.5, "Routine execution context", and trigger execution contexts in Subclause 4.38.2, "Trigger execution".

4.37.5 Routine execution context

A routine execution context consists of:

— An indication as to whether or not an SQL-invoked routine is active.

— An SQL-data access indication, which identifies what SQL-statements, if any, are allowed during the execution of an SQL-invoked routine. The SQL-data access indication is one of the following: does not possibly contain SQL, possibly contains SQL, possibly reads SQL-data, or possibly modifies SQL-data.

— An identification of the SQL-invoked routine that is active.

— The routine SQL-path derived from the routine SQL-path if the SQL-invoked routine that is active is an SQL routine and from the external routine SQL-path if the SQL-invoked routine that is active is an external routine.

An SQL-invoked routine is active as soon as an SQL-statement executed by an SQL-agent causes invocation of an SQL-invoked routine and ceases to be active when execution of that invocation is complete.

When an SQL-agent causes the invocation of an SQL-invoked routine, a new context for the current SQL-session is created and the values of the current context are preserved. When the execution of that SQL-invoked routine completes, the original context of the current SQL-session is restored and some SQL-session characteristics are reset.

If the routine execution context of the SQL-session indicates that an SQL-invoked routine is active, then the routine SQL-path included in the routine execution context of the SQL-session is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in a <preparable statement> or in a <direct SQL statement>.

4.38 Triggers

4.38.1 General description of triggers

A trigger is a specification for a given action to take place every time a given operation takes place on a given object. The action, known as a *triggered action*, is an SQL-procedure statement or a list of such statements. The object is a persistent base table known as the *subject table* of the trigger. The operation, known as a *trigger event*, is either deletion, insertion, or replacement of a collection of rows.

The triggered action is specified to take place either immediately before the triggering event or immediately after it, according to its specified *trigger action time*, BEFORE or AFTER. The trigger is a *BEFORE trigger* or an *AFTER trigger*, according to its trigger action time.

A trigger is either a *delete trigger*, an *insert trigger*, or an *update trigger*, according to the nature of its trigger event.

Every trigger event arises as a consequence of executing some SQL-data change statement. That consequence might be direct, as for example when the SQL-data change statement is an <insert statement> operating on a base table, or indirect, as for example in the following cases:

— The SQL-data change statement is a <merge statement>.

— The SQL-data change statement operates on the referenced table of some foreign key whose referential action is CASCADE, SET NULL, or SET DEFAULT.

— The SQL-data change statement operates on a viewed table.

A triggered action is permitted to include SQL-data change statements that give rise to trigger events.

A collection of rows being deleted, inserted or replaced is known as a *transition table*. For a delete trigger there is just one transition table, known as an *old transition table*. For an insert trigger there is just one transition table, known as a *new transition table*. For an update trigger there is both an old transition table (the rows being replaced) and a new transition table (the replacement rows), these two tables having the same cardinality.

A reference to “the transition table” of a trigger is ambiguous in the case of an update trigger but whenever such a reference appears in this International Standard it is immaterial to which of the two transition tables it applies.

The triggered action can be specified to take place either just once when the trigger event takes place, in which case the trigger is a *statement-level trigger*, or once for each row of the transition table when the trigger event takes place, in which case the trigger is a *row-level trigger*.

If the triggered action is specified to take place before the event, the trigger is a row-level trigger, and there is a new transition table, then the action can include statements whose effect is to alter the effect of the impending operation.

Special variables make the data in the transition table(s) available to the triggered action. For a statement-level trigger the variable is one whose value is a transition table. For a row-level trigger, the variable is a range variable, known as a *transition variable*. A transition variable ranges over the rows of a transition table, each row giving rise to exactly one execution of the triggered action, with the row in question assigned to the transition variable. A transition variable is either an *old transition variable* or a *new transition variable*, depending on the transition table over whose rows it ranges.

When there are two transition tables, old and new, each row in the new transition table is one that is derived by an update operation applied to exactly one row in the old transition table. Thus there is a 1:1 correspondence between the rows of the two tables. However, this correspondence is visible only to a row-level trigger, each invocation of which is able to access both the old and new transition variables, the new transition variable representing the result of applying the update operation in question to the row in the old transition variable.

A trigger is defined by a <trigger definition>, specifying the name of the trigger, its subject table, its trigger event, its trigger action time, whether it is statement-level or row-level, names as required for referencing transition tables or variables, and the triggered action.

A schema might include one or more trigger descriptors, each of which includes a triggered action specifying a <triggered SQL statement> that is to be executed (either once for each affected row, in the case of a row-level trigger, or once for the whole trigger event in the case of a statement-level trigger) immediately before or immediately after the trigger event takes place. The execution of a triggered action might cause the triggering of further triggered actions. It does so if it entails execution of an SQL-procedure statement whose effect causes the trigger event of some trigger to take place.

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

— The name of the trigger.

— The name of the subject table.

— The trigger action time (BEFORE or AFTER).

— The trigger event (INSERT, DELETE, or UPDATE).

— Whether the trigger is a statement-level trigger or a row-level trigger.

— Any old transition variable name, new transition variable name, old transition table name, or new transition table name.

— The triggered action.

— The trigger column list (possibly empty) for the trigger event.

— The triggered action column set of the triggered action.

— The timestamp of creation of the trigger.

The *order of execution* of a set of triggers is ascending by value of their timestamp of creation in their descriptors, such that the oldest trigger executes first. If one or more triggers have the same timestamp value, then their relative order of execution is implementation-defined.

A triggered action is always executed under the authorization of the owner of the schema that includes the trigger.

4.38.2 Trigger execution

During the execution of an SQL-statement *S*, zero or more *trigger execution contexts* exist, no more than one of which is *active* at any one time. A trigger execution context TEC_i comes into existence, becomes the active one, ceases to be active, and is destroyed as and when required under the General Rules for *S*.

If, while TEC_i is active, the General Rules for *S* require some new trigger execution context TEC_j to come into existence, then TEC_j replaces TEC_i as the active trigger execution context. TEC_i becomes active again when TEC_j is destroyed.

Multiple trigger execution contexts exist when the General Rules for *S* specify the execution of another SQL-procedure statement *T* before the execution of *S* is complete, and the General Rules for *T* require a new trigger execution context to come into existence.

A trigger execution context consists of a set of *state changes*. Within a trigger execution context, each state change is uniquely identified by a trigger event, a subject table, and a *column list*. The trigger event can be DELETE, INSERT, or UPDATE.

A state change *SC* consists of:

— A *set of transitions*.

— A trigger event.

— A subject table.

— A column list.

— A set (initially empty) of statement-level triggers *considered as executed* for *SC*.

— A set of row-level triggers, each paired with the set of rows in *SC* for which it is considered as executed.

What constitutes a transition depends on the trigger event. If the trigger event is DELETE, a transition is a row in the old transition table. If the trigger event is INSERT, a transition is a row in the new transition table. If the trigger event is UPDATE, a transition is a row *OR* in the old transition table paired with a row *NR* in the new transition table, such that *NR* is the row derived by applying a specified update operation to *OR*. *OR* and *NR* are the *old row* and the *new row*, respectively, of the transition.

A statement-level trigger that is considered as executed for a state change *SC* (in a given trigger execution context) is not subsequently executed for *SC*.

If a row-level trigger *RLT* is considered as executed for some row *R* in *SC*, then *RLT* is not subsequently executed for *R*.

A consequence of the execution of an SQL-data change statement is called an *SQL-update operation* if and only if that consequence causes at least one transition to arise in some state change.

A (possibly empty) old transition table exists if the trigger event is UPDATE or DELETE. It consists of a copy of each row that is to be updated in or deleted from the subject table. A (possibly empty) new transition table exists if the trigger event is UPDATE or INSERT. It consists of a copy of each row that results from updating a row in the subject table or is to be inserted into the subject table.

A <triggered action> may refer to the old transition table only if an <old transition table name> is specified for it in the <trigger definition>, and to the new transition table only if a <new transition table name> is specified for it in the <trigger definition>.

The <triggered action> of a row-level trigger may refer to a range variable ranging over the rows of the old transition table only if an <old transition variable name> is specified for it in the <trigger definition>. Similarly, the <triggered action> of a row-level trigger may refer to a range variable ranging over the rows of the new transition table only if a <new transition variable name> is specified for it in the <trigger definition>. The scope of a transition variable or transition table name is the <triggered action> of the <trigger definition> that specifies it, excluding any <SQL schema statement>s that are contained in that <triggered action>.

When execution of an SQL-data change statement causes a trigger execution context TEC_i to come into existence, the set of state changes SSC_i in TEC_i is empty. Let $SC_{i,j}$ be a state change in SSC_i. Let *TE* be the trigger event (DELETE, INSERT, or UPDATE) of $SC_{i,j}$. Let *ST* be the subject table of $SC_{i,j}$.

If *TE* is INSERT or DELETE, then let *PSC* be a set whose only element is the empty set.

If *TE* is UPDATE, then:

— Let *CL* be the list of columns being updated by SSC_i.

— Let *OC* be the set of column names identifying the columns in *CL*.

— Let *PSC* be the set consisting of the empty set and every subset of the set of column names of *ST* that has at least one column that is in *OC*.

Let *PSCN* be the number of elements in *PSC*. A state change $SC_{i,j}$, for *j* varying from 1 (one) to *PSCN*, identified by *TE*, *ST*, and the *j*-th element in *PSC*, is added to SSC_i, provided that SSC_i does not already contain a state change corresponding to $SC_{i,j}$. Transitions are added to $SC_{i,j}$ as specified by the General Rules of Subclause 15.7, "Effect of deleting rows from base tables", Subclause 15.10, "Effect of inserting tables into base tables", and Subclause 15.13, "Effect of replacing rows in base tables".

When a state change $SC_{i,j}$ arises in SSC_i, one or more triggers are *activated by* $SC_{i,j}$. A trigger *TR* is activated by $SC_{i,j}$ if and only if the subject table of *TR* is the subject table of $SC_{i,j}$, the trigger event of *TR* is the trigger

event of $SC_{i,j}$, and the set of column names listed in the trigger column list of *TR* is equivalent to the set of column names listed in $SC_{i,j}$.

NOTE 68 — The trigger column list is included in the descriptor of *TR*; it is empty if the trigger event is DELETE or INSERT. The trigger column list is also empty if the trigger event is UPDATE, but the <trigger event> of the <trigger definition> that defined *TR* does not specify a <trigger column list>.

For each state change $SC_{i,j}$ in TEC_i, the BEFORE triggers activated by $SC_{i,j}$ are executed before any of their triggering events take effect. When those triggering events have taken effect, any AFTER triggers activated by the state changes of TEC_i are executed.

The <triggered action> contained in a <trigger definition> for a BEFORE or AFTER row-level trigger can refer to columns of old transition variables and new transition variables. Such references can be specified as <column reference>s, which can be <target specification>s and <simple target specification>s when they refer to columns of the new transition variable.

NOTE 69 — By using such <column reference>s as <assignment target>s (see ISO/IEC 9075-4), the triggered action of a BEFORE trigger is able to cause certain SQL-data change statements to have different effects from those specified in the statements.

When an execution of the <triggered SQL statement> *TSS* of a triggered action is not successful, then an exception condition is raised and the SQL-statement that caused *TSS* to be executed has no effect on SQL-data or schemas.

4.39 Client-server operation

This Subclause is modified by Subclause 4.5, "Client-server operation", in ISO/IEC 9075-3.

When an SQL-agent is active, it is bound in some implementation-defined manner to a single SQL-client. That SQL-client processes the explicit or implicit <SQL connection statement> for the first call to an externally-invoked procedure by an SQL-agent. The SQL-client communicates with, either directly or possibly through other agents such as RDA, one or more SQL-servers. An SQL-session involves an SQL-agent, an SQL-client, and a single SQL-server.

SQL-client modules associated with the SQL-agent exist in the SQL-environment containing the SQL-client associated with the SQL-agent.

Called <externally-invoked procedure>s and <direct SQL statement>s containing an <SQL connection statement> or an <SQL diagnostics statement> are processed by the SQL-client. Following the successful execution of a <connect statement> or a <set connection statement>, the SQL-client modules associated with the SQL-agent are effectively materialized with an implementation-dependent <SQL-client module name> in the SQL-server. Other called <externally-invoked procedure>s and <direct SQL statement>s are processed by the SQL-server.

A call by the SQL-agent to an <externally-invoked procedure> whose <SQL procedure statement> simply contains an <SQL diagnostics statement> fetches information from the specified diagnostics area in the diagnostics area stack associated with the SQL-client. Following the execution of an <SQL procedure statement> by an SQL-server, diagnostic information is passed in an implementation-dependent manner into the SQL-agent's diagnostics area stack in the SQL-client. The effect on diagnostic information of incompatibilities between the character repertoires supported by the SQL-client and SQL-server is implementation-dependent.

(Blank page)

5 Lexical elements

This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-4.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-9.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-10.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-13.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-14.

5.1 <SQL terminal character>

This Subclause is modified by Subclause 5.1, "<SQL terminal character>", in ISO/IEC 9075-10.

Function

Define the terminal symbols of the SQL language and the elements of strings.

Format

```
<SQL terminal character> ::=
  <SQL language character>

<SQL language character> ::=
    <simple Latin letter>
  | <digit>
  | <SQL special character>

<simple Latin letter> ::=
    <simple Latin upper case letter>
  | <simple Latin lower case letter>

<simple Latin upper case letter> ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
  | P | Q | R | S | T | U | V | W | X | Y | Z

<simple Latin lower case letter> ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
  | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<SQL special character> ::=
    <space>
  | <double quote>
  | <percent>
  | <ampersand>
  | <quote>
```

```
  | <left paren>
  | <right paren>
  | <asterisk>
  | <plus sign>
  | <comma>
  | <minus sign>
  | <period>
  | <solidus>
  | <colon>
  | <semicolon>
  | <less than operator>
  | <equals operator>
  | <greater than operator>
  | <question mark>
  | <left bracket>
  | <right bracket>
  | <circumflex>
  | <underscore>
  | <vertical bar>
  | <left brace>
  | <right brace>

<space> ::=
  !! See the Syntax Rules

<double quote> ::=
  "

<percent> ::=
  %

<ampersand> ::=
  &

<quote> ::=
  '

<left paren> ::=
  (

<right paren> ::=
  )

<asterisk> ::=
  *

<plus sign> ::=
  +

<comma> ::=
  ,

<minus sign> ::=
  -

<period> ::=
  .

<solidus> ::=
```

```
  /

<reverse solidus> ::=
  \

<colon> ::=
  :

<semicolon> ::=
  ;

<less than operator> ::=
  <

<equals operator> ::=
  =

<greater than operator> ::=
  >

<question mark> ::=
  ?

<left bracket or trigraph> ::=
    <left bracket>
  | <left bracket trigraph>

<right bracket or trigraph> ::=
    <right bracket>
  | <right bracket trigraph>

<left bracket> ::=
  [

<left bracket trigraph> ::=
  ??(

<right bracket> ::=
  ]

<right bracket trigraph> ::=
  ??)

<circumflex> ::=
  ^

<underscore> ::=
  _

<vertical bar> ::=
  |

<left brace> ::=
  {

<right brace> ::=
  }
```

Syntax Rules

1) Every character set shall contain a <space> character that is equivalent to U+0020.

Access Rules

None.

General Rules

1) There is a one-to-one correspondence between the symbols contained in <simple Latin upper case letter> and the symbols contained in <simple Latin lower case letter> such that, for all *i*, the symbol defined as the *i*-th alternative for <simple Latin upper case letter> corresponds to the symbol defined as the *i*-th alternative for <simple Latin lower case letter>.

Conformance Rules

None.

5.2 <token> and <separator>

This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 5.2, "<token> and <separator>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-14.

Function

Specify lexical units (tokens and separators) that participate in SQL language.

Format

```
<token> ::=
    <nondelimiter token>
  | <delimiter token>

<nondelimiter token> ::=
    <regular identifier>
  | <key word>
  | <unsigned numeric literal>
  | <national character string literal>
  | <binary string literal>
  | <large object length token>
  | <Unicode delimited identifier>
  | <Unicode character string literal>
  | <SQL language identifier>

<regular identifier> ::=
  <identifier body>

<identifier body> ::=
  <identifier start> [ <identifier part>... ]

<identifier part> ::=
    <identifier start>
  | <identifier extend>

<identifier start> ::=
  !! See the Syntax Rules

<identifier extend> ::=
  !! See the Syntax Rules

<large object length token> ::=
  <digit>... <multiplier>

<multiplier> ::=
    K
  | M
  | G
  | T
  | P
```

```
<delimited identifier> ::=
  <double quote> <delimited identifier body> <double quote>

<delimited identifier body> ::=
  <delimited identifier part>...

<delimited identifier part> ::=
    <nondoublequote character>
  | <doublequote symbol>

<Unicode delimited identifier> ::=
  U<ampersand><double quote> <Unicode delimiter body> <double quote>
      <Unicode escape specifier>

<Unicode escape specifier> ::=
  [ UESCAPE <quote><Unicode escape character><quote> ]

<Unicode delimiter body> ::=
  <Unicode identifier part>...

<Unicode identifier part> ::=
    <delimited identifier part>
  | <Unicode escape value>

<Unicode escape value> ::=
    <Unicode 4 digit escape value>
  | <Unicode 6 digit escape value>
  | <Unicode character escape value>

<Unicode 4 digit escape value> ::=
  <Unicode escape character><hexit><hexit><hexit><hexit>

<Unicode 6 digit escape value> ::=
  <Unicode escape character><plus sign>
      <hexit><hexit><hexit><hexit><hexit><hexit>

<Unicode character escape value> ::=
  <Unicode escape character><Unicode escape character>

<Unicode escape character> ::=
  !! See the Syntax Rules

<nondoublequote character> ::=
  !! See the Syntax Rules

<doublequote symbol> ::=
  "" !! two consecutive double quote characters

<delimiter token> ::=
    <character string literal>
  | <date string>
  | <time string>
  | <timestamp string>
  | <interval string>
  | <delimited identifier>
  | <SQL special character>
  | <not equals operator>
  | <greater than or equals operator>
  | <less than or equals operator>
  | <concatenation operator>
```

```
  | <right arrow>
  | <left bracket trigraph>
  | <right bracket trigraph>
  | <double colon>
  | <double period>

<not equals operator> ::=
  <>

<greater than or equals operator> ::=
  >=

<less than or equals operator> ::=
  <=

<concatenation operator> ::=
  ||

<right arrow> ::=
  ->

<double colon> ::=
  ::

<double period> ::=
  ..

<separator> ::=
  { <comment> | <white space> }...

<white space> ::=
  !! See the Syntax Rules

<comment> ::=
    <simple comment>
  | <bracketed comment>

<simple comment> ::=
  <simple comment introducer> [ <comment character>... ] <newline>

<simple comment introducer> ::=
  <minus sign><minus sign>

<bracketed comment> ::=
  <bracketed comment introducer>
      <bracketed comment contents>
      <bracketed comment terminator>

<bracketed comment introducer> ::=
  /*

<bracketed comment terminator> ::=
  */

<bracketed comment contents> ::=
  !! See the Syntax Rules
      [ { <comment character> | <separator> }... ]

<comment character> ::=
    <nonquote character>
```

```
  | <quote>

<newline> ::=
  !! See the Syntax Rules

<key word> ::=
    <reserved word>
  | <non-reserved word>

<non-reserved word> ::=
    A | ABSOLUTE | ACTION | ADA | ADD | ADMIN | AFTER | ALWAYS | ASC
  | ASSERTION | ASSIGNMENT | ATTRIBUTE | ATTRIBUTES

  | BEFORE | BERNOULLI | BREADTH

  | C | CASCADE | CATALOG | CATALOG_NAME | CHAIN | CHARACTER_SET_CATALOG
  | CHARACTER_SET_NAME | CHARACTER_SET_SCHEMA | CHARACTERISTICS | CHARACTERS
  | CLASS_ORIGIN | COBOL | COLLATION | COLLATION_CATALOG | COLLATION_NAME | COLLATION_SCHEMA
  | COLUMN_NAME | COMMAND_FUNCTION | COMMAND_FUNCTION_CODE | COMMITTED
  | CONDITION_NUMBER | CONNECTION | CONNECTION_NAME | CONSTRAINT_CATALOG | CONSTRAINT_NAME
  | CONSTRAINT_SCHEMA | CONSTRAINTS | CONSTRUCTOR | CONTAINS | CONTINUE | CURSOR_NAME

  | DATA | DATETIME_INTERVAL_CODE | DATETIME_INTERVAL_PRECISION | DEFAULTS | DEFERRABLE
  | DEFERRED | DEFINED | DEFINER | DEGREE | DEPTH | DERIVED | DESC | DESCRIPTOR
  | DIAGNOSTICS | DISPATCH | DOMAIN | DYNAMIC_FUNCTION | DYNAMIC_FUNCTION_CODE

  | EQUALS | EXCLUDE | EXCLUDING

  | FINAL | FIRST | FLAG | FOLLOWING | FORTRAN | FOUND

  | G | GENERAL | GENERATED | GO | GOTO | GRANTED

  | HIERARCHY

  | IMMEDIATE | IMPLEMENTATION | INCLUDING | INCREMENT | INITIALLY | INPUT | INSTANCE
  | INSTANTIABLE | INVOKER | ISOLATION

  | K | KEY | KEY_MEMBER | KEY_TYPE

  | LAST | LENGTH | LEVEL | LOCATOR

  | M | MAP | MATCHED | MAXVALUE | MESSAGE_LENGTH | MESSAGE_OCTET_LENGTH
  | MESSAGE_TEXT | MINVALUE | MORE | MUMPS

  | NAME | NAMES | NESTING | NEXT | NFC | NFD | NFKC | NFKD
  | NORMALIZED | NULLABLE | NULLS | NUMBER

  | OBJECT | OCTETS | OPTION | OPTIONS | ORDERING | ORDINALITY | OTHERS
  | OUTPUT | OVERRIDING

  | P | PAD | PARAMETER_MODE | PARAMETER_NAME | PARAMETER_ORDINAL_POSITION
  | PARAMETER_SPECIFIC_CATALOG | PARAMETER_SPECIFIC_NAME | PARAMETER_SPECIFIC_SCHEMA
  | PARTIAL | PASCAL | PATH | PLACING | PLI | PRECEDING | PRESERVE | PRIOR
  | PRIVILEGES | PUBLIC

  | READ | RELATIVE | REPEATABLE | RESTART | RESTRICT | RETURNED_CARDINALITY
  | RETURNED_LENGTH | RETURNED_OCTET_LENGTH | RETURNED_SQLSTATE | ROLE
  | ROUTINE | ROUTINE_CATALOG | ROUTINE_NAME | ROUTINE_SCHEMA | ROW_COUNT

  | SCALE | SCHEMA | SCHEMA_NAME | SCOPE_CATALOG | SCOPE_NAME | SCOPE_SCHEMA
```

```
  | SECTION | SECURITY | SELF | SEQUENCE | SERIALIZABLE | SERVER_NAME | SESSION
  | SETS | SIMPLE | SIZE | SOURCE | SPACE | SPECIFIC_NAME | STATE | STATEMENT
  | STRUCTURE | STYLE | SUBCLASS_ORIGIN

  | T | TABLE_NAME | TEMPORARY | TIES | TOP_LEVEL_COUNT | TRANSACTION
  | TRANSACTION_ACTIVE | TRANSACTIONS_COMMITTED | TRANSACTIONS_ROLLED_BACK
  | TRANSFORM | TRANSFORMS | TRIGGER_CATALOG | TRIGGER_NAME | TRIGGER_SCHEMA | TYPE

  | UNBOUNDED | UNCOMMITTED | UNDER | UNNAMED | USAGE | USER_DEFINED_TYPE_CATALOG
  | USER_DEFINED_TYPE_CODE | USER_DEFINED_TYPE_NAME | USER_DEFINED_TYPE_SCHEMA

  | VIEW

  | WORK | WRITE

  | ZONE

<reserved word> ::=
    ABS | ALL | ALLOCATE | ALTER | AND | ANY | ARE | ARRAY | AS | ASENSITIVE
  | ASYMMETRIC | AT | ATOMIC | AUTHORIZATION | AVG

  | BEGIN | BETWEEN | BIGINT | BINARY | BLOB | BOOLEAN | BOTH | BY

  | CALL | CALLED | CARDINALITY | CASCADED | CASE | CAST | CEIL | CEILING
  | CHAR | CHAR_LENGTH | CHARACTER | CHARACTER_LENGTH | CHECK | CLOB | CLOSE
  | COALESCE | COLLATE | COLLECT | COLUMN | COMMIT | CONDITION | CONNECT
  | CONSTRAINT | CONVERT | CORR | CORRESPONDING | COUNT | COVAR_POP | COVAR_SAMP
  | CREATE | CROSS | CUBE | CUME_DIST | CURRENT | CURRENT_CATALOG | CURRENT_DATE
  | CURRENT_DEFAULT_TRANSFORM_GROUP | CURRENT_PATH | CURRENT_ROLE | CURRENT_SCHEMA
  | CURRENT_TIME | CURRENT_TIMESTAMP | CURRENT_TRANSFORM_GROUP_FOR_TYPE
  | CURRENT_USER | CURSOR | CYCLE

  | DATE | DAY | DEALLOCATE | DEC | DECIMAL | DECLARE | DEFAULT | DELETE
  | DENSE_RANK | DEREF | DESCRIBE | DETERMINISTIC | DISCONNECT | DISTINCT
  | DOUBLE | DROP | DYNAMIC

  | EACH | ELEMENT | ELSE | END | END-EXEC | ESCAPE | EVERY | EXCEPT | EXEC
  | EXECUTE | EXISTS | EXP | EXTERNAL | EXTRACT

  | FALSE | FETCH | FILTER | FLOAT | FLOOR | FOR | FOREIGN | FREE | FROM
  | FULL | FUNCTION | FUSION

  | GET | GLOBAL | GRANT | GROUP | GROUPING

  | HAVING | HOLD | HOUR

  | IDENTITY | IN | INDICATOR | INNER | INOUT | INSENSITIVE | INSERT
  | INT | INTEGER | INTERSECT | INTERSECTION | INTERVAL | INTO | IS

  | JOIN

  | LANGUAGE | LARGE | LATERAL | LEADING | LEFT | LIKE | LIKE_REGEX
  | LN | LOCAL | LOCALTIME | LOCALTIMESTAMP | LOWER

  | MATCH | MAX | MEMBER | MERGE | METHOD | MIN | MINUTE
  | MOD | MODIFIES | MODULE | MONTH | MULTISET

  | NATIONAL | NATURAL | NCHAR | NCLOB | NEW | NO | NONE | NORMALIZE | NOT
  | NULL | NULLIF | NUMERIC
```

```
| OCTET_LENGTH | OCCURRENCES_REGEX | OF | OLD | ON | ONLY | OPEN | OR
| ORDER | OUT | OUTER | OVER | OVERLAPS | OVERLAY

| PARAMETER | PARTITION | PERCENT_RANK | PERCENTILE_CONT | PERCENTILE_DISC
| POSITION | POSITION_REGEX | POWER | PRECISION | PREPARE | PRIMARY
| PROCEDURE

| RANGE | RANK | READS | REAL | RECURSIVE | REF | REFERENCES | REFERENCING
| REGR_AVGX | REGR_AVGY | REGR_COUNT | REGR_INTERCEPT | REGR_R2 | REGR_SLOPE
| REGR_SXX | REGR_SXY | REGR_SYY | RELEASE | RESULT | RETURN | RETURNS
| REVOKE | RIGHT | ROLLBACK | ROLLUP | ROW | ROW_NUMBER | ROWS

| SAVEPOINT | SCOPE | SCROLL | SEARCH | SECOND | SELECT | SENSITIVE
| SESSION_USER | SET | SIMILAR | SMALLINT | SOME | SPECIFIC | SPECIFICTYPE
| SQL | SQLEXCEPTION | SQLSTATE | SQLWARNING | SQRT | START | STATIC
| STDDEV_POP | STDDEV_SAMP | SUBMULTISET | SUBSTRING | SUBSTRING_REGEX | SUM
| SYMMETRIC | SYSTEM | SYSTEM_USER

| TABLE | TABLESAMPLE | THEN | TIME | TIMESTAMP | TIMEZONE_HOUR | TIMEZONE_MINUTE
| TO | TRAILING | TRANSLATE | TRANSLATE_REGEX | TRANSLATION | TREAT
| TRIGGER | TRIM | TRUE

| UESCAPE | UNION | UNIQUE | UNKNOWN | UNNEST | UPDATE  | UPPER | USER | USING

| VALUE | VALUES | VAR_POP | VAR_SAMP | VARBINARY | VARCHAR | VARYING

| WHEN | WHENEVER | WHERE | WIDTH_BUCKET | WINDOW | WITH | WITHIN | WITHOUT

| YEAR
```

Syntax Rules

1) An <identifier start> is any character in the Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, or “Nl”.

   NOTE 70 — The Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, and “Nl” are assigned to Unicode characters that are, respectively, upper-case letters, lower-case letters, title-case letters, modifier letters, other letters, and letter numbers.

2) An <identifier extend> is U+00B7, “Middle Dot”, or any character in the Unicode General Category classes “Mn”, “Mc”, “Nd”, “Pc”, or “Cf”.

   NOTE 71 — The Unicode General Category classes “Mn”, “Mc”, “Nd”, “Pc”, and “Cf” are assigned to Unicode characters that are, respectively, nonspacing marks, spacing combining marks, decimal numbers, connector punctuations, and formatting codes.

3) <white space> is any consecutive sequence of characters each of which satisfies the definition of white space found in Subclause 3.1.6, “Definitions provided in Part 2”.

4) <newline> is the implementation-defined end-of-line indicator.

   NOTE 72 — <newline> is typically represented by U+000A (“Line Feed”) and/or U+000D (“Carriage Return”); however, this representation is not required by ISO/IEC 9075.

5) With the exception of the <space> character explicitly contained in <binary string literal>, <timestamp string>, and <interval string>, a <token>, other than a <character string literal>, a <national character string literal>, a <Unicode character string literal>, a <delimited identifier>, or a <Unicode delimited identifier> shall not contain a <space> character or other <separator>.

6) A <nondoublequote character> is any character of the source language character set other than a <double quote>.

NOTE 73 — “source language character set” is defined in Subclause 4.8.1, “Host languages”, in ISO/IEC 9075-1.

7) Any <token> may be followed by a <separator>. A <nondelimiter token> shall be followed by a <delimiter token> or a <separator>.

NOTE 74 — If the Format does not allow a <nondelimiter token> to be followed by a <delimiter token>, then that <nondelimiter token> shall be followed by a <separator>.

8) There shall be no <separator> separating the <minus sign>s of a <simple comment introducer>.

9) There shall be no <separator> separating any two <digit>s or separating a <digit> and <multiplier> of a <large object length token>.

10) Within a <bracketed comment contents>, any <solidus> immediately followed by an <asterisk> without any intervening <separator> shall be considered to be the <bracketed comment introducer> of a <separator> that is a <bracketed comment>.

NOTE 75 — Conforming programs should not place <simple comment> within a <bracketed comment> because if such a <simple comment> contains the sequence of characters “*/” without a preceding “/*” in the same <simple comment>, it will prematurely terminate the containing <bracketed comment>.

11) SQL text containing one or more instances of <comment> is equivalent to the same SQL text with the <comment> replaced with <newline>.

12) In a <regular identifier>, the number of <identifier part>s shall be less than 128.

13) The <delimited identifier body> of a <delimited identifier> shall not comprise more than 128 <delimited identifier part>s.

14) In a <Unicode delimited identifier>, there shall be no <separator> between the 'U' and the <ampersand> nor between the <ampersand> and the <double quote>.

15) In a <Unicode delimited identifier>, the introductory 'U' may be represented either in upper case (as 'U') or in lower case (as 'u').

16) <Unicode escape character> shall be a single character from the source language character set other than a <hexit>, <plus sign>, <quote>, <double quote>, or <white space>.

17) If the source language character set contains <reverse solidus>, then let *DEC* be <reverse solidus>; otherwise, let *DEC* be an implementation-defined character from the source language character set that is not a <hexit>, <plus sign>, <quote>, <double quote>, or <white space>.

18) If a <Unicode escape specifier> does not contain <Unicode escape character>, then “UESCAPE <quote>*DEC*<quote>” is implicit.

19) In a <Unicode escape value> there shall be no <separator> between the <Unicode escape character> and the first <hexit>, nor between any of the <hexit>s.

20) The <Unicode delimiter body> of a <Unicode delimited identifier> shall not comprise more than 128 <Unicode identifier part>s.

21) <Unicode 4 digit escape value> '<Unicode escape character>xyzw' is equivalent to the Unicode code point specified by U+xyzw.

22) <Unicode 6 digit escape value> '<Unicode escape character>+xyzwrs' is equivalent to the character at the Unicode code point specified by U+xyzwrs.

NOTE 76 — The 6-hexit notation is derived by taking the UCS-4 notation defined in ISO/IEC 10646-1 and removing the leading two hexits, whose values are always 0 (zero).

23) <Unicode character escape value> is equivalent to a single instance of <Unicode escape character>.

24) For every <identifier body> *IB* there is exactly one corresponding case-normal form *CNF*. *CNF* is an <identifier body> derived from *IB* as follows:

Let *n* be the number of characters in *IB*. For *i* ranging from 1 (one) to *n*, the *i*-th character M_i of *IB* is transliterated into the corresponding character or characters of *CNF* as follows:

Case:

a) If M_i is a lower case character or a title case character for which an equivalent upper case sequence *U* is defined by Unicode, then let *j* be the number of characters in *U*; the next *j* characters of *CNF* are *U*.

b) Otherwise, the next character of *CNF* is M_i.

25) The case-normal form of the <identifier body> of a <regular identifier> is used for purposes such as and including determination of identifier equivalence, representation in the Definition and Information Schemas, and representation in diagnostics areas.

NOTE 77 — The Information Schema and Definition Schema are defined in ISO/IEC 9075-11.

NOTE 78 — Any lower-case letters for which there are no upper-case equivalents are left in their lower-case form.

26) The case-normal form of <regular identifier> shall not be equal, according to the comparison rules in Subclause 8.2, "<comparison predicate>", to any <reserved word> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER.

27) Two <regular identifier>s are equivalent if the case-normal forms of their <identifier body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation *IDC* that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

28) A <regular identifier> and a <delimited identifier> are equivalent if the case-normal form of the <identifier body> of the <regular identifier> and the <delimited identifier body> of the <delimited identifier> (with all occurrences of <quote> replaced by <quote symbol> and all occurrences of <doublequote symbol> replaced by <double quote>), considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and *IDC*, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

29) Two <delimited identifier>s are equivalent if their <delimited identifier body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

30) Two <Unicode delimited identifier>s are equivalent if their <Unicode delimiter body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

31) A <Unicode delimited identifier> and a <delimited identifier> are equivalent if their <Unicode delimiter body> and <delimited identifier body>, respectively, each considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

32) For the purposes of identifying <key word>s, any <simple Latin lower case letter> contained in a candidate <key word> shall be effectively treated as the corresponding <simple Latin upper case letter>.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F391, “Long identifiers”, in a <regular identifier>, the number of <identifier part>s shall be less than 18.

2) Without Feature F391, “Long identifiers”, the <delimited identifier body> of a <delimited identifier> shall not comprise more than 18 <delimited identifier part>s.

NOTE 79 — Not every character set supported by a conforming SQL-implementation necessarily contains every character associated with <identifier start> and <identifier part> that is identified in the Syntax Rules of this Subclause. No conforming SQL-implementation shall be required to support in <identifier start> or <identifier part> any character identified in the Syntax Rules of this Subclause unless that character belongs to the character set in use for an SQL-client module or in SQL-data.

3) Without Feature T351, “Bracketed comments”, conforming SQL language shall not contain a <bracketed comment>.

4) Without Feature F392, “Unicode escapes in identifiers”, conforming SQL language shall not contain a <Unicode delimited identifier>.

5) Without Feature T043, “Multiplier T”, in conforming SQL language, a <multiplier> shall not be T.

6) Without Feature T044, “Multiplier P”, in conforming SQL language, a <multiplier> shall not be P.

5.3 <literal>

Function

Specify a non-null value.

Format

```
<literal> ::=
    <signed numeric literal>
  | <general literal>

<unsigned literal> ::=
    <unsigned numeric literal>
  | <general literal>

<general literal> ::=
    <character string literal>
  | <national character string literal>
  | <Unicode character string literal>
  | <binary string literal>
  | <datetime literal>
  | <interval literal>
  | <boolean literal>

<character string literal> ::=
  [ <introducer><character set specification> ]
      <quote> [ <character representation>... ] <quote>
      [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<introducer> ::=
  <underscore>

<character representation> ::=
    <nonquote character>
  | <quote symbol>

<nonquote character> ::=
  !! See the Syntax Rules.

<quote symbol> ::=
  <quote><quote>

<national character string literal> ::=
  N <quote> [ <character representation>... ]
      <quote> [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<Unicode character string literal> ::=
  [ <introducer><character set specification> ]
      U<ampersand><quote> [ <Unicode representation>... ] <quote>
      [ { <separator> <quote> [ <Unicode representation>... ] <quote> }... ]
      <Unicode escape specifier>

<Unicode representation> ::=
    <character representation>
  | <Unicode escape value>
```

```
<binary string literal> ::=
  X <quote> [ <space>... ] [ { <hexit> [ <space>... ] <hexit> [ <space>... ] }... ] <quote>
      [ { <separator> <quote> [ <space>... ] [ { <hexit> [ <space>... ]
      <hexit> [ <space>... ] }... ] <quote> }... ]

<hexit> ::=
    <digit> | A | B | C | D | E | F | a | b | c | d | e | f

<signed numeric literal> ::=
  [ <sign> ] <unsigned numeric literal>

<unsigned numeric literal> ::=
    <exact numeric literal>
  | <approximate numeric literal>

<exact numeric literal> ::=
    <unsigned integer> [ <period> [ <unsigned integer> ] ]
  | <period> <unsigned integer>

<sign> ::=
    <plus sign>
  | <minus sign>

<approximate numeric literal> ::=
  <mantissa> E <exponent>

<mantissa> ::=
  <exact numeric literal>

<exponent> ::=
  <signed integer>

<signed integer> ::=
  [ <sign> ] <unsigned integer>

<unsigned integer> ::=
  <digit>...

<datetime literal> ::=
    <date literal>
  | <time literal>
  | <timestamp literal>

<date literal> ::=
  DATE <date string>

<time literal> ::=
  TIME <time string>

<timestamp literal> ::=
  TIMESTAMP <timestamp string>

<date string> ::=
  <quote> <unquoted date string> <quote>

<time string> ::=
  <quote> <unquoted time string> <quote>

<timestamp string> ::=
  <quote> <unquoted timestamp string> <quote>
```

```
<time zone interval> ::=
  <sign> <hours value> <colon> <minutes value>

<date value> ::=
  <years value> <minus sign> <months value> <minus sign> <days value>

<time value> ::=
  <hours value> <colon> <minutes value> <colon> <seconds value>

<interval literal> ::=
  INTERVAL [ <sign> ] <interval string> <interval qualifier>

<interval string> ::=
  <quote> <unquoted interval string> <quote>

<unquoted date string> ::=
  <date value>

<unquoted time string> ::=
  <time value> [ <time zone interval> ]

<unquoted timestamp string> ::=
  <unquoted date string> <space> <unquoted time string>

<unquoted interval string> ::=
  [ <sign> ] { <year-month literal> | <day-time literal> }

<year-month literal> ::=
    <years value> [ <minus sign> <months value> ]
  | <months value>

<day-time literal> ::=
    <day-time interval>
  | <time interval>

<day-time interval> ::=
  <days value> [ <space> <hours value> [ <colon> <minutes value>
      [ <colon> <seconds value> ] ] ]

<time interval> ::=
    <hours value> [ <colon> <minutes value> [ <colon> <seconds value> ] ]
  | <minutes value> [ <colon> <seconds value> ]
  | <seconds value>

<years value> ::=
  <datetime value>

<months value> ::=
  <datetime value>

<days value> ::=
  <datetime value>

<hours value> ::=
  <datetime value>

<minutes value> ::=
  <datetime value>

<seconds value> ::=
```

```
  <seconds integer value> [ <period> [ <seconds fraction> ] ]

<seconds integer value> ::=
  <unsigned integer>

<seconds fraction> ::=
  <unsigned integer>

<datetime value> ::=
  <unsigned integer>

<boolean literal> ::=
    TRUE
  | FALSE
  | UNKNOWN
```

Syntax Rules

1) In a <character string literal> or <national character string literal>, the sequence:

```
<quote> <character representation>... <quote><separator>
<quote> <character representation>... <quote>
```

is equivalent to the sequence

```
<quote> <character representation>... <character representation>... <quote>
```

NOTE 80 — The <character representation>s in the equivalent sequence are in the same sequence and relative sequence as in the original <character string literal>.

2) In a <Unicode character string literal>, the sequence:

```
<quote> <Unicode representation>... <quote>
<separator> <quote> <Unicode representation>... <quote>
```

is equivalent to the sequence:

```
<quote> <Unicode representation>... <Unicode representation>... <quote>
```

3) In a <Unicode character string literal>, the introductory 'U' may be represented either in upper case (as 'U') or in lower case (as 'u').

4) In a <binary string literal>, the sequence

```
<quote> [ <space>... ] { <hexit> [ <space>... ]
<hexit> [ <space>... ] }... <quote>
```

is equivalent to the sequence

```
<quote> { <hexit> <hexit> }... <quote>
```

NOTE 81 — The <hexit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <binary string literal>.

5) In a <binary string literal>, the sequence

```
<quote> { <hexit> <hexit> }... <quote> <separator>
<quote> { <hexit> <hexit> }... <quote>
```

is equivalent to the sequence

```
<quote> { <hexit> <hexit> }... { <hexit> <hexit> }... <quote>
```

NOTE 82 — The <hexit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <binary string literal>.

6) In a <binary string literal>, the introductory 'X' may be represented either in upper case (as 'X') or in lower case (as 'x').

7) In a <character string literal>, <national character string literal>, <Unicode character string literal>, or <binary string literal>, a <separator> shall contain a <newline>.

8) A <national character string literal> is equivalent to a <character string literal> with the “N” replaced by “<introducer><character set specification>”, where “<character set specification>” is an implementation-defined <character set name>.

9) In a <national character string literal>, the introductory 'N' may be represented either in upper case (as 'N') or in lower case (as 'n').

10) In a <Unicode character string literal> that specifies “<introducer><character set specification>”, there shall be no <separator> between the <introducer> and the <character set specification>.

11) In a <Unicode character string literal>, there shall be no <separator> between the “U” and the <ampersand> nor between the <ampersand> and the <quote>.

12) The character set of a <Unicode character string literal> that specifies “<introducer><character set specification>” is the character set specified by the <character set specification>. The character set of a <Unicode character string literal> that does not specify “<introducer><character set specification>” is the character set of the SQL-client module that contains the <Unicode character string literal>.

13) A <Unicode character string literal> is equivalent to a <character string literal> in which every <Unicode escape value> has been replaced with the equivalent Unicode character. The set of characters contained in the <Unicode character string literal> shall be wholly contained in the character set of the <Unicode character string literal>.

NOTE 83 — The requirement for “wholly contained” applies after the replacement of <Unicode escape value>s with equivalent Unicode characters.

14) Each <character representation> is a character of the source language character set. The value of a <character string literal>, viewed as a string in the source language character set, shall be equivalent to a character string of the implicit or explicit character set of the <character string literal> or <national character string literal>.

NOTE 84 — “source language character set” is defined in Subclause 4.8.1, “Host languages”, in ISO/IEC 9075-1.

15) A <nonquote character> is one of:

   a) Any character of the source language character set other than a <quote>.

   b) Any character other than a <quote> in the character set identified by the <character set specification> or implied by “N”.

16) Case:

   a) If a <character set specification> is not specified in a <character string literal>, then the set of characters contained in the <character string literal> shall be wholly contained in the character set of the <SQL-client module definition> that contains the <character string literal>.

b) Otherwise, there shall be no <separator> between the <introducer> and the <character set specification>, and the set of characters contained in the <character string literal> shall be wholly contained in the character set specified by the <character set specification>.

17) The declared type of a <character string literal> is fixed-length character string. The length of a <character string literal> is the number of <character representation>s that it contains. Each <quote symbol> contained in <character string literal> represents a single <quote> in both the value and the length of the <character string literal>. The two <quote>s contained in a <quote symbol> shall not be separated by any <separator>.

NOTE 85 — <character string literal>s are allowed to be zero-length strings (*i.e.*, to contain no characters) even though it is not permitted to declare a <data type> that is CHARACTER with <length> 0 (zero).

18) The character set of a <character string literal> is

Case:

a) If the <character string literal> specifies a <character set specification>, then the character set specified by that <character set specification>.

b) Otherwise, the character set of the SQL-client module that contains the <character string literal>.

19) The declared type collation of a <character string literal> is the character set collation, and the collation derivation is *implicit*.

20) It is implementation-defined whether the declared type of a <binary string literal> is a fixed-length binary string type, a variable-length binary string type, or a binary large object string type. Each <hexit> appearing in the literal is equivalent to a quartet of bits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are interpreted as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111, respectively. The <hexit>s a, b, c, d, e, and f have respectively the same values as the <hexit>s A, B, C, D, E, and F.

21) An <exact numeric literal> without a <period> has an implied <period> following the last <digit>.

22) The declared type of an <exact numeric literal> *ENL* is an implementation-defined exact numeric type whose scale is the number of <digit>s to the right of the <period>. There shall be an exact numeric type capable of representing the value of *ENL* exactly.

23) The declared type of an <approximate numeric literal> *ANL* is an implementation-defined approximate numeric type. The value of *ANL* shall not be greater than the maximum value nor less than the minimum value that can be represented by the approximate numeric types.

NOTE 86 — Thus the only syntax error for an <approximate numeric literal> is what is commonly known as “overflow”; there is no syntax error for specifying more significant digits than the SQL-implementation can represent internally, nor for specifying a value that has no exact equivalent in the SQL-implementation's internal representation. (“Underflow”, *i.e.*, specifying a nonzero value so close to 0 (zero) that the closest representation in the SQL-implementation's internal representation is 0E0, is a special case of the latter condition, and is not a syntax error.)

24) In an <approximate numeric literal>, the exponent indicator 'E' may be represented either in upper case (as 'E') or in lower case (as 'e').

25) The declared type of a <date literal> is DATE.

26) The declared type of a <time literal> that does not specify <time zone interval> is TIME(*P*) WITHOUT TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise. The declared type of a <time literal> that specifies <time zone interval> is TIME(*P*) WITH TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise.

27) The declared type of a <timestamp literal> that does not specify <time zone interval> is TIMESTAMP(*P*) WITHOUT TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise. The declared type of a <timestamp literal> that specifies <time zone interval> is TIMESTAMP(*P*) WITH TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise.

28) If <time zone interval> is not specified, then the effective <time zone interval> of the datetime data type is the current default time zone displacement for the SQL-session.

29) Let *datetime component* be either <years value>, <months value>, <days value>, <hours value>, <minutes value>, or <seconds value>.

30) Let *N* be the number of <primary datetime field>s in the precision of the <interval literal>, as specified by <interval qualifier>.

    The <interval literal> being defined shall contain *N* datetime components.

    The declared type of <interval literal> specified with an <interval qualifier> is INTERVAL with the <interval qualifier>.

    Each datetime component shall have the precision specified by the <interval qualifier>.

31) Within a <datetime literal>, the <years value> shall contain four digits. The <seconds integer value> and other datetime components, with the exception of <seconds fraction>, shall each contain two digits.

32) Within the definition of a <datetime literal>, the <datetime value>s are constrained by the natural rules for dates and times according to the Gregorian calendar.

33) Within the definition of an <interval literal>, the <datetime value>s are constrained by the natural rules for intervals according to the Gregorian calendar.

34) Within the definition of an <interval literal> that contains a <year-month literal>, the <interval qualifier> shall not specify DAY, HOUR, MINUTE, or SECOND. Within the definition of an <interval literal> that contains a <day-time literal>, the <interval qualifier> shall not specify YEAR or MONTH.

35) Within the definition of a <datetime literal>, the value of the <time zone interval> shall be in the range –12:59 to +14:00.

Access Rules

None.

General Rules

1) The value of a <character string literal> is the result of transliterating the sequence of <character representation>s that it contains from the source language character set to the implicit or explicit character set of the <character string literal>.

2) If the character repertoire of a <character string literal> *US* is UCS, then its value is replaced by NORMALIZE(*US*).

3) The numeric value of an <exact numeric literal> is determined by the normal mathematical interpretation of positional decimal notation.

4) The numeric value of an <approximate numeric literal> is approximately the product of the exact numeric value represented by the <mantissa> with the number obtained by raising the number 10 to the power of the exact numeric value represented by the <exponent>.

5) The <sign> in a <signed numeric literal> or an <interval literal> is a monadic arithmetic operator. The monadic arithmetic operators + and – specify monadic plus and monadic minus, respectively. If neither monadic plus nor monadic minus are specified in a <signed numeric literal> or an <interval literal>, or if monadic plus is specified, then the literal is positive. If monadic minus is specified in a <signed numeric literal> or <interval literal>, then the literal is negative. If <sign> is specified in both possible locations in an <interval literal>, then the sign of the literal is determined by normal mathematical interpretation of multiple sign operators.

6) Let *V* be the integer value of the <unsigned integer> contained in <seconds fraction> and let *N* be the number of digits in the <seconds fraction> respectively. The resultant value of the <seconds fraction> is effectively determined as follows.

   Case:

   a) If <seconds fraction> is specified within the definition of a <datetime literal>, then the effective value of the <seconds fraction> is $V * 10^{-N}$ seconds.

   b) If <seconds fraction> is specified within the definition of an <interval literal>, then let *M* be the <interval fractional seconds precision> specified in the <interval qualifier>.

      Case:

      i) If $N < M$, then let *V1* be $V * 10^{M-N}$; the effective value of the <seconds fraction> is $V1 * 10^{-M}$ seconds.

      ii) If $N > M$, then let *V2* be the integer part of the quotient of $V/10^{N-M}$; the effective value of the <seconds fraction> is $V2 * 10^{-M}$ seconds.

      iii) Otherwise, the effective value of the <seconds fraction> is $V * 10^{-M}$ seconds.

7) The *i*-th datetime component in a <datetime literal> or <interval literal> assigns the value of the datetime component to the *i*-th <primary datetime field> in the <datetime literal> or <interval literal>.

8) If <time zone interval> is specified, then the time and timestamp values in <time literal> and <timestamp literal> represent a datetime in the specified time zone.

9) If <date value> is specified, then it is interpreted as a date in the Gregorian calendar. If <time value> is specified, then it is interpreted as a time of day. Let *DV* be the value of the <datetime literal>, disregarding <time zone interval>.

   Case:

   a) If <time zone interval> is specified, then let *TZI* be the value of the interval denoted by <time zone interval>. The value of the <datetime literal> is *DV* – *TZI*, with time zone displacement *TZI*.

   b) Otherwise, the value of the <datetime literal> is *DV*.

NOTE 87 — If <time zone interval> is specified, then a <time literal> or <timestamp literal> is interpreted as local time with the specified time zone displacement. However, it is effectively converted to UTC while retaining the original time zone displacement.

If <time zone interval> is not specified, then no assumption is made about time zone displacement. However, should a time zone displacement be required during subsequent processing, the current default time zone displacement of the SQL-session will be applied at that time.

10) The truth value of a <boolean literal> is *True* if TRUE is specified, is *False* if FALSE is specified, and is *Unknown* if UNKNOWN is specified.

NOTE 88 — The null value of the boolean data type is equivalent to the *Unknown* truth value (see Subclause 4.5, "Boolean types").

Conformance Rules

1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean literal>.

2) Without Feature F555, "Enhanced seconds precision", in conforming SQL language, an <unsigned integer> that is a <seconds fraction> that is contained in a <timestamp literal> shall not contain more than 6 <digit>s.

3) Without Feature F555, "Enhanced seconds precision", in conforming SQL language, a <time literal> shall not contain a <seconds fraction>.

4) Without Feature F421, "National character", conforming SQL language shall not contain a <national character string literal>.

5) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval literal>.

6) Without Feature F271, "Compound character literals", in conforming SQL language, a <character string literal> shall contain exactly one repetition of <character representation> (that is, it shall contain exactly one sequence of "<quote> <character representation>... <quote>").

7) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <time zone interval>.

8) Without Feature T041, "Basic LOB data type support", or Feature T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain a <binary string literal>.

9) Without Feature T023, "Compound binary literals", in conforming SQL language, a <binary string literal> shall contain exactly one repetition of "<quote> [{ <hexit> <hexit> }...] <quote>".

10) Without Feature T024, "Spaces in binary literals", in conforming SQL language, a <binary string literal> shall not contain a <space>.

11) Without Feature F393, "Unicode escapes in literals", conforming SQL language shall not contain a <Unicode character string literal>.

5.4 Names and identifiers

This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-14.

Function

Specify names.

Format

```
<identifier> ::=
  <actual identifier>

<actual identifier> ::=
    <regular identifier>
  | <delimited identifier>
  | <Unicode delimited identifier>

<SQL language identifier> ::=
  <SQL language identifier start> [ <SQL language identifier part>... ]

<SQL language identifier start> ::=
  <simple Latin letter>

<SQL language identifier part> ::=
    <simple Latin letter>
  | <digit>
  | <underscore>

<authorization identifier> ::=
    <role name>
  | <user identifier>

<table name> ::=
  <local or schema qualified name>

<domain name> ::=
  <schema qualified name>

<schema name> ::=
  [ <catalog name> <period> ] <unqualified schema name>

<unqualified schema name> ::=
  <identifier>

<catalog name> ::=
  <identifier>

<schema qualified name> ::=
  [ <schema name> <period> ] <qualified identifier>

<local or schema qualified name> ::=
```

```
  [ <local or schema qualifier> <period> ] <qualified identifier>

<local or schema qualifier> ::=
    <schema name>
  | <local qualifier>

<qualified identifier> ::=
  <identifier>

<column name> ::=
  <identifier>

<correlation name> ::=
  <identifier>

<query name> ::=
  <identifier>

<SQL-client module name> ::=
  <identifier>

<procedure name> ::=
  <identifier>

<schema qualified routine name> ::=
  <schema qualified name>

<method name> ::=
  <identifier>

<specific name> ::=
  <schema qualified name>

<cursor name> ::=
  <local qualified name>

<local qualified name> ::=
  [ <local qualifier> <period> ] <qualified identifier>

<local qualifier> ::=
  MODULE

<host parameter name> ::=
  <colon> <identifier>

<SQL parameter name> ::=
  <identifier>

<constraint name> ::=
  <schema qualified name>

<external routine name> ::=
    <identifier>
  | <character string literal>

<trigger name> ::=
  <schema qualified name>

<collation name> ::=
  <schema qualified name>
```

```
<character set name> ::=
  [ <schema name> <period> ] <SQL language identifier>

<transliteration name> ::=
  <schema qualified name>

<transcoding name> ::=
  <schema qualified name>

<schema-resolved user-defined type name> ::=
  <user-defined type name>

<user-defined type name> ::=
  [ <schema name> <period> ] <qualified identifier>

<attribute name> ::=
  <identifier>

<field name> ::=
  <identifier>

<savepoint name> ::=
  <identifier>

<sequence generator name> ::=
  <schema qualified name>

<role name> ::=
  <identifier>

<user identifier> ::=
  <identifier>

<connection name> ::=
  <simple value specification>

<SQL-server name> ::=
  <simple value specification>

<connection user name> ::=
  <simple value specification>

<SQL statement name> ::=
    <statement name>
  | <extended statement name>

<statement name> ::=
  <identifier>

<extended statement name> ::=
  [ <scope option> ] <simple value specification>

<dynamic cursor name> ::=
    <cursor name>
  | <extended cursor name>

<extended cursor name> ::=
  [ <scope option> ] <simple value specification>

<descriptor name> ::=
```

```
  [ <scope option> ] <simple value specification>

<scope option> ::=
    GLOBAL
  | LOCAL

<window name> ::=
  <identifier>
```

Syntax Rules

1) In an <SQL language identifier>, the number of <SQL language identifier part>s shall be less than 128.

2) An <SQL language identifier> is equivalent to an <SQL language identifier> in which every letter that is a lower-case letter is replaced by the corresponding upper-case letter or letters. This treatment includes determination of equivalence, representation in the Information and Definition Schemas, representation in diagnostics areas, and similar uses.

   NOTE 89 — The Information Schema and Definition Schema are defined in ISO/IEC 9075-11.

3) An <SQL language identifier> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER, shall not be equal, according to the comparison rules in Subclause 8.2, "<comparison predicate>", to any <reserved word> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER.

   NOTE 90 — It is the intention that no <key word> specified in ISO/IEC 9075 or revisions thereto shall end with an <underscore>.

4) If a <local or schema qualified name> does not contain a <local or schema qualifier>, then

   Case:

   a) If the <local or schema qualified name> is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

   b) If the <local or schema qualified name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

   c) Otherwise, the <schema name> that is specified or implicit for the SQL-client module is implicit.

5) Let *TN* be a <table name> with a <qualified identifier> *QI* and a <local or schema qualifier> *LSQ*.

   Case:

   a) If *LSQ* is "MODULE", then *TN* shall be contained in an <SQL-client module definition> *M* and the <module contents> of *M* shall contain a <temporary table declaration> *TT* whose <table name> has a <qualified identifier> equivalent to *QI*.

   b) Otherwise, *LSQ* shall be a <schema name> that identifies a schema that contains a <table definition> or <view definition> whose <table name> has a <qualified identifier> equivalent to *QI*.

6) If a <cursor name> *CN* with a <qualified identifier> *QI* does not contain a <local qualifier>, then the <local qualifier> MODULE is implicit.

7) Let *CN* be a <cursor name>. *CN* shall be contained, without an intervening <SQL schema statement>, in an <SQL-client module definition> whose <module contents> contain a <declare cursor> or <dynamic declare cursor> whose <cursor name> is *CN*.

8) If <user-defined type name> *UDTN* with a <qualified identifier> *QI* is specified, then

   Case:

   a) If *UDTN* is simply contained in <path-resolved user-defined type name>, then

      Case:

      i) If *UDTN* contains a <schema name> *SN*, then the schema identified by *SN* shall contain the descriptor of a user-defined type *UDT* such that the <qualified identifier> of *UDT* is equivalent to *QI*. *UDT* is the user-defined type identified by *UDTN*.

      ii) Otherwise,

         1) Case:

            A) If *UDTN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then let *DP* be the SQL-path of the current SQL-session.

            B) If *UDTN* is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

            C) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains *UDTN*.

         2) Let *N* be the number of <schema name>s in *DP*. Let S_i, 1 (one) $\leq i \leq N$, be the *i*-th <schema name> in *DP*.

         3) Let the *set of subject types* be the set containing every user-defined type *T* in the schema identified by some S_i, 1 (one) $\leq i \leq N$, such that the <qualified identifier> of *T* is equivalent to *QI*. There shall be at least one type in the set of subject types.

         4) Let *UDT* be the user-defined type contained in the set of subject types such that there is no other type *UDT2* for which the <schema name> of the schema that includes the user-defined type descriptor of *UDT2* precedes in *DP* the <schema name> identifying the schema that includes the user-defined type descriptor of *UDT*. *UDTN* identifies *UDT*.

         5) The implicit <schema name> of *UDTN* is the <schema name> of the schema that includes the user-defined type descriptor of *UDT*.

   b) If *UDTN* is simply contained in <schema-resolved user-defined type name> and *UDTN* does not contain a <schema name>, then

      Case:

      i) If *UDTN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the implicit <schema name> of *UDTN* is the default <unqualified schema name> for the SQL-session.

ii) If *UDTN* is contained in a <schema definition>, then the implicit <schema name> of *UDTN* is the <schema name> that is specified or implicit in <schema definition>.

iii) Otherwise, the implicit <schema name> of *UDTN* is the <schema name> that is specified or implicit in <SQL-client module definition>.

9) Two <user-defined type name>s are equivalent if and only if they have equivalent <qualified identifier>s and equivalent <schema name>s, regardless of whether the <schema name>s are implicit or explicit.

10) No <unqualified schema name> shall specify DEFINITION_SCHEMA.

11) If a <transcoding name> does not specify a <schema name>, then INFORMATION_SCHEMA is implicit; otherwise, INFORMATION_SCHEMA shall be specified.

12) If a <character set name> does not specify a <schema name>, then

Case:

a) If <character set name> is not immediately contained in:

i) A <character set definition>.

ii) A <drop character set statement>.

then <schema name> INFORMATION_SCHEMA is implicit.

b) Otherwise,

Case:

i) If the <character set name> is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

ii) If the <character set name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

iii) Otherwise, the <character set name> that is specified or implicit for the <SQL-client module definition> is implicit.

13) If a <schema qualified name> *SQN* other than a <transcoding name> does not contain a <schema name>, then

Case:

a) If any of the following is true:

i) *SQN* is immediately contained in a <collation name> that is not immediately contained in a <collation definition> or in a <drop collation statement>.

ii) *SQN* is immediately contained in a <transliteration name> that is not immediately contained in a <transliteration definition> or in a <drop transliteration statement>.

then<schema name> INFORMATION_SCHEMA is implicit.

b) If *SQN* is immediately contained in a <constraint name> that is contained in a <table definition> or an <alter table statement>, then the explicit or implicit <schema name> of the <table name> of the table identified by the <table definition> or <alter table statement> is implicit.

c) If *SQN* is immediately contained in a <constraint name> that is contained in a <domain definition> or an <alter domain statement>, then the explicit or implicit <schema name> of the the <domain name> of the domain identified by the <domain definition> or an <alter domain statement> is implicit.

d) Otherwise,

Case:

i) If *SQN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

ii) If *SQN* is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

iii) Otherwise, the <schema name> that is specified or implicit for the <SQL-client module definition> is implicit.

14) If a <schema name> does not contain a <catalog name>, then

Case:

a) If the <unqualified schema name> is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default catalog name for the SQL-session is implicit.

b) If the <unqualified schema name> is contained in a <module authorization clause>, then an implementation-defined <catalog name> is implicit.

c) If the <unqualified schema name> is contained in a <schema definition> other than in a <schema name clause>, then the <catalog name> that is specified or implicit in the <schema name clause> is implicit.

d) If the <unqualified schema name> is contained in a <schema name clause>, then

Case:

i) If the <schema name clause> is contained in an <SQL-client module definition>, then the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.

ii) Otherwise, an implementation-defined <catalog name> is implicit.

e) Otherwise, the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.

15) Two <schema qualified name>s are equivalent if and only if their <qualified identifier>s are equivalent and their <schema name>s are equivalent, regardless of whether the <schema name>s are implicit or explicit.

16) Two <local or schema qualified name>s are equivalent if and only if their <qualified identifier>s are equivalent and either they both specify MODULE or they both specify or imply <schema name>s that are equivalent.

17) Two <character set name>s are equivalent if and only if their <SQL language identifier>s are equivalent and their <schema name>s are equivalent, regardless of whether the <schema name>s are implicit or explicit.

18) Two <schema name>s are equivalent if and only if their <unqualified schema name>s are equivalent and their <catalog name>s are equivalent, regardless of whether the <catalog name>s are implicit or explicit.

19) An <identifier> that is a <correlation name> is associated with a table within a particular scope.

NOTE 91 — The scope of a <correlation name> is defined in the Syntax Rules of Subclause 7.6, "<table reference>", Subclause 11.39, "<trigger definition>", and elsewhere. Scopes may be nested. In different scopes, <correlation name>s that are equivalent <identifier>s may be associated with different tables or with the same table.

20) No <authorization identifier> shall specify "PUBLIC".

21) Those <identifier>s that are valid <authorization identifier>s are implementation-defined.

22) Those <identifier>s that are valid <catalog name>s are implementation-defined.

23) The <data type> of <SQL-server name>, <connection name>, and <connection user name> shall be character string with an implementation-defined character set and shall have an octet length of 128 characters or less.

24) The <simple value specification> of <extended statement name> or <extended cursor name> shall not be a <literal>.

25) If an <extended statement name> or <extended cursor name> contains an <SQL parameter reference>, then it shall also contain an explicit <scope option>.

26) The declared type of the <simple value specification> of <extended statement name> shall be character string with an implementation-defined character set and shall have an octet length of 128 octets or less.

27) The declared type of the <simple value specification> of <extended cursor name> shall be character string with an implementation-defined character set and shall have an octet length of 128 octets or less.

28) The declared type of the <simple value specification> of <descriptor name> shall be character string with an implementation-defined character set and shall have an octet length of 128 octets or less.

29) In a <descriptor name>, <extended statement name>, or <extended cursor name>, if a <scope option> is not specified, then a <scope option> of LOCAL is implicit. If a <scope option> is contained in an <SQL schema statement> then it shall not contain LOCAL.

Access Rules

None.

General Rules

1) A <table name> identifies a table.

2) Within its scope, a <correlation name> identifies a table.

3) Within its scope, a <query name> identifies the table defined or returned by some associated <query expression body>.

4) A <column name> identifies a column.

5) A <domain name> identifies a domain.

6) An <authorization identifier> identifies a set of privileges.

7) An <SQL-client module name> identifies an SQL-client module.

8) A <schema qualified routine name> identifies an SQL-invoked routine.

9) A <method name> identifies an SQL-invoked method *M* whose descriptor is included in the schema that includes the descriptor of the user-defined type that is the type of *M*.

10) A <specific name> identifies an SQL-invoked routine.

11) A <cursor name> identifies a cursor.

12) A <host parameter name> identifies a host parameter.

13) An <SQL parameter name> identifies an SQL parameter.

14) An <external routine name> identifies an external routine.

15) A <trigger name> identifies a trigger.

16) A <constraint name> identifies a table constraint, a domain constraint, or an assertion.

17) A <catalog name> identifies a catalog.

18) A <schema name> identifies a schema.

19) A <collation name> identifies a collation.

20) A <character set name> identifies a character set.

21) A <transliteration name> identifies a character transliteration.

22) A <transcoding name> identifies a transcoding. All <transcoding name>s are implementation-defined.

23) A <connection name> identifies an SQL-connection.

24) A <user-defined type name> identifies a user-defined type.

25) An <attribute name> identifies an attribute of a structured type.

26) A <savepoint name> identifies a savepoint. The scope of a <savepoint name> is the SQL-transaction in which it was defined.

27) A <sequence generator name> identifies a sequence generator.

28) A <field name> identifies a field.

29) A <role name> identifies a role.

30) A <user identifier> identifies a user.

31) If a prepared statement *PSX* is created in SQL-session *SS* by executing a <prepare statement> *PS1* that contains an <extended statement name> *ESN1* whose value at the time of execution is *V*, then, for as long as it exists, *PSX* can be identified by an <extended statement name> *ESN2* in an <SQL procedure statement> *PS2* executed in *SS* if the value of *ESN2* at the time of execution is *V* and the <scope option> of *ESN2* is the same as the <scope option> of *ESN1*. If the <scope option> of *ESN1* is LOCAL, then *ESN2* identifies *PSX* only if *PS2* is contained in the same <SQL-client module definition> as *PS1*.

NOTE 92 — The "value at the time of execution" is defined in the General Rules of Subclause 20.6, "<prepare statement>".

32) Case:

a) A <dynamic cursor name> that is a <cursor name> is a non-extended name that identifies a declared dynamic cursor in an <SQL dynamic statement>.

b) If a <dynamic cursor name> is an <extended cursor name>, then the value of the <simple value specification> simply contained in the <extended cursor name> is an extended name that identifies an extended dynamic cursor in an <SQL dynamic statement>.

NOTE 93 — The scope of a non-extended name is defined in Subclause 4.24.2, "Dynamic SQL statements and descriptor areas".

33) A <statement name> is a non-extended name that identifies a prepared statement created by the execution of a <prepare statement>.

34) If an extended dynamic cursor *CSR* is created in SQL-session *SS* by executing an <allocate cursor statement> *ACS* that contains an <extended statement name> *ESN1* whose value at the time of execution is *V*, then, for as long as it exists, *CSR* can be identified by an <extended statement name> *ESN2* in an <SQL procedure statement> *PS2* executed in *SS* if the value of *ESN2* at the time of execution is *V* and the <scope option> of *ESN2* is the same as the <scope option> of *ESN1*. If the <scope option> of *ESN1* is LOCAL, then *ESN2* identifies *CSR* only if *PS2* is contained in the same <SQL-client module definition> as *ACS*.

NOTE 94 — The "value at the time of execution" is defined in the General Rules of Subclause 20.15, "<allocate cursor statement>".

35) If an SQL descriptor area *SDA* is created in SQL-session *SS* by executing an <allocate descriptor statement> *ADS* that contains an <extended statement name> *ESN1* whose value at the time of execution is *V*, then, for as long as it exists, *SDA* can be identified by an <extended statement name> *ESN2* in an <SQL procedure statement> *PS2* executed in *SS* if the value of *ESN2* at the time of execution is *V* and the <scope option> of *ESN2* is the same as the <scope option> of *ESN1*. If the <scope option> of *ESN1* is LOCAL, then *ESN2* identifies *SDA* only if *PS2* is contained in the same <SQL-client module definition> as *ADS*.

NOTE 95 — The "value at the time of execution" is defined in the General Rules of Subclause 20.2, "<allocate descriptor statement>".

36) A <window name> identifies a window.

Conformance Rules

1) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <savepoint name>.

2) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <role name>.

3) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", conforming SQL language shall not contain a <query name>.

4) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <attribute name>.

5) Without Feature T051, "Row types", conforming SQL language shall not contain a <field name>.

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

7) Without Feature F771, "Connection management", conforming SQL language shall not contain an explicit <connection name>.

8) Without Feature F690, "Collation support", conforming SQL language shall not contain a <collation name>.

9) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transliteration name>.

10) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transcoding name>.

11) Without Feature F821, “Local table references”, conforming SQL language shall not contain a <local or schema qualifier> that contains a <local qualifier>.

12) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <domain name>.

13) Without Feature F491, “Constraint management”, conforming SQL language shall not contain a <constraint name>.

14) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <character set name>.

15) Without Feature T601, “Local cursor references”, a <cursor name> shall not contain a <local qualifier>.

16) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <extended statement name> or <extended cursor name>.

17) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain an <SQL statement name>.

18) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain <dynamic cursor name>.

19) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <descriptor name>.

20) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window name>.

21) Without Feature T176, “Sequence generator support”, conforming SQL language shall not contain a <sequence generator name>.

22) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <descriptor name> that is not a <literal>.

(Blank page)

6 Scalar expressions

This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-4.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-9.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-10.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-13.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-14.

6.1 <data type>

This Subclause is modified by Subclause 6.1, "<data type>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.1, "<data type>", in ISO/IEC 9075-14.

Function

Specify a data type.

Format

```
<data type> ::=
    <predefined type>
  | <row type>
  | <path-resolved user-defined type name>
  | <reference type>
  | <collection type>

<predefined type> ::=
    <character string type> [ CHARACTER SET <character set specification> ]
        [ <collate clause> ]
  | <national character string type> [ <collate clause> ]
  | <binary string type>
  | <numeric type>
  | <boolean type>
  | <datetime type>
  | <interval type>

<character string type> ::=
    CHARACTER [ <left paren> <length> <right paren> ]
  | CHAR [ <left paren> <length> <right paren> ]
  | CHARACTER VARYING <left paren> <length> <right paren>
  | CHAR VARYING <left paren> <length> <right paren>
  | VARCHAR <left paren> <length> <right paren>
  | <character large object type>

<character large object type> ::=
    CHARACTER LARGE OBJECT [ <left paren> <large object length> <right paren> ]
```

```
  | CHAR LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | CLOB [ <left paren> <large object length> <right paren> ]

<national character string type> ::=
    NATIONAL CHARACTER [ <left paren> <length> <right paren> ]
  | NATIONAL CHAR [ <left paren> <length> <right paren> ]
  | NCHAR [ <left paren> <length> <right paren> ]
  | NATIONAL CHARACTER VARYING <left paren> <length> <right paren>
  | NATIONAL CHAR VARYING <left paren> <length> <right paren>
  | NCHAR VARYING <left paren> <length> <right paren>
  | <national character large object type>

<national character large object type> ::=
    NATIONAL CHARACTER LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | NCHAR LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | NCLOB [ <left paren> <large object length> <right paren> ]

<binary string type> ::=
    BINARY [ <left paren> <length> <right paren> ]
  | BINARY VARYING <left paren> <length> <right paren>
  | VARBINARY <left paren> <length> <right paren>
  | <binary large object string type>

<binary large object string type> ::=
    BINARY LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | BLOB [ <left paren> <large object length> <right paren> ]

<numeric type> ::=
    <exact numeric type>
  | <approximate numeric type>

<exact numeric type> ::=
    NUMERIC [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | DECIMAL [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | DEC [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | SMALLINT
  | INTEGER
  | INT
  | BIGINT

<approximate numeric type> ::=
    FLOAT [ <left paren> <precision> <right paren> ]
  | REAL
  | DOUBLE PRECISION

<length> ::=
  <unsigned integer> [ <char length units> ]

<large object length> ::=
    <unsigned integer> [ <multiplier> ] [ <char length units> ]
  | <large object length token> [ <char length units> ]

<char length units> ::=
    CHARACTERS
  | OCTETS

<precision> ::=
  <unsigned integer>
```

```
<scale> ::=
  <unsigned integer>

<boolean type> ::=
  BOOLEAN

<datetime type> ::=
    DATE
  | TIME [ <left paren> <time precision> <right paren> ] [ <with or without time zone> ]
  | TIMESTAMP [ <left paren> <timestamp precision> <right paren> ]
      [ <with or without time zone> ]

<with or without time zone> ::=
    WITH TIME ZONE
  | WITHOUT TIME ZONE

<time precision> ::=
  <time fractional seconds precision>

<timestamp precision> ::=
  <time fractional seconds precision>

<time fractional seconds precision> ::=
  <unsigned integer>

<interval type> ::=
  INTERVAL <interval qualifier>

<row type> ::=
  ROW <row type body>

<row type body> ::=
  <left paren> <field definition> [ { <comma> <field definition> }... ] <right paren>

<reference type> ::=
  REF <left paren> <referenced type> <right paren> [ <scope clause> ]

<scope clause> ::=
  SCOPE <table name>

<referenced type> ::=
  <path-resolved user-defined type name>

<path-resolved user-defined type name> ::=
  <user-defined type name>

<collection type> ::=
    <array type>
  | <multiset type>

<array type> ::=
  <data type> ARRAY
      [ <left bracket or trigraph> <maximum cardinality> <right bracket or trigraph> ]

<maximum cardinality> ::=
  <unsigned integer>

<multiset type> ::=
  <data type> MULTISET
```

Syntax Rules

1) CHAR is equivalent to CHARACTER. DEC is equivalent to DECIMAL. INT is equivalent to INTEGER. VARCHAR is equivalent to CHARACTER VARYING. NCHAR is equivalent to NATIONAL CHARACTER. CLOB is equivalent to CHARACTER LARGE OBJECT. NCLOB is equivalent to NATIONAL CHARACTER LARGE OBJECT. VARBINARY is equivalent to BINARY VARYING. BLOB is equivalent to BINARY LARGE OBJECT.

2) "NATIONAL CHARACTER" is equivalent to the corresponding <character string type> with a specification of "CHARACTER SET *CSN*", where "*CSN*" is an implementation-defined <character set name>.

3) If <character string type> is specified, then the collation derivation of the resulting character string type is *implicit*.

   Case:

   a) If <collate clause> is specified, then the collation specified by it shall be applicable to the explicit or implicit character set *CS* of the character string type. That collation is the declared type collation of the character string type.

   b) Otherwise, the character set collation of *CS* is the declared type collation of the character string type.

4) The value of a <length> or a <precision> shall be greater than 0 (zero).

5) If <length> is omitted, then a <length> of 1 (one) is implicit.

6) A <binary large object string type> shall not contain a <char length units>.

7) If <char length units> is specified, then the character repertoire of the explicit or implicit character set of the character string type shall be UCS.

8) If <data type> contains a <character string type> or <character large object type> and does not contain <char length units>, then CHARACTERS is implicit.

9) If <large object length> is omitted, then an implementation-defined <large object length> is implicit.

10) The numeric value of a <large object length> is determined as follows.

    Case:

    a) If <large object length> immediately contains <unsigned integer> and does not immediately contain <multiplier>, then the numeric value of <large object length> is the numeric value of the specified <unsigned integer>.

    b) If <large object length> immediately contains <large object length token> or immediately contains <unsigned integer> and <multiplier>, then let *D* be the value of the specified <unsigned integer> or the numeric value of the sequence of <digit>s of <large object length token> interpreted as an <unsigned integer>. The numeric value of <large object length> is the numeric value resulting from the multiplication of *D* and *MS*, where *MS* is:

       i) If <multiplier> is K, then 1,024.

       ii) If <multiplier> is M, then 1,048,576.

       iii) If <multiplier> is G, then 1,073,741,824.

       iv) If <multiplier> is T, then 1,099,511,627,776.

v) If <multiplier> is P, then 1,125,899,906,842,624.

11) If a <scale> is omitted, then a <scale> of 0 (zero) is implicit.

12) If a <precision> is omitted, then an implementation-defined <precision> is implicit.

13) CHARACTER specifies the data type character string.

14) Characters in a character string are numbered beginning with 1 (one).

15) Case:

a) If neither VARYING nor LARGE OBJECT is specified in <character string type>, then the length in characters of the character string is fixed and is the value of <length>.

b) If VARYING is specified in <character string type>, then the length in characters of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <length>.

c) If LARGE OBJECT is specified in a <character string type>, then the length in characters of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.

16) The maximum values of <length> and <large object length> are implementation-defined. Neither <length> nor <large object length> shall be greater than the corresponding maximum value.

17) If <character string type> is not contained in a <domain definition> or a <column definition> and CHARACTER SET is not specified, then an implementation-defined <character set specification> that specifies an implementation-defined character set that contains at least every character that is in <SQL language character> is implicit.

NOTE 96 — Subclause 11.24, "<domain definition>", and Subclause 11.4, "<column definition>", specify the result when <character string type> is contained in a <domain definition> or <column definition>, respectively.

18) BINARY specifies the data type binary string.

19) Octets in a binary string are numbered beginning with 1 (one).

20) Case:

a) If neither VARYING nor LARGE OBJECT is specified in a <binary string type>, then the length in octets of the binary string is fixed and is the value of <length>.

b) If VARYING is specified in a <binary string type>, then the length in octets of the binary string is variable, with a minimum of length of 0 (zero) and a maximum length of the value of <length>.

c) If LARGE OBJECT is specified in a <binary string type>, then the length in octets of the binary string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.

21) The <scale> of an <exact numeric type> shall not be greater than the <precision> of the <exact numeric type>.

22) For the <exact numeric type>s DECIMAL and NUMERIC:

a) The maximum value of <precision> is implementation-defined. <precision> shall not be greater than this value.

b) The maximum value of <scale> is implementation-defined. <scale> shall not be greater than this maximum value.

23) NUMERIC specifies the data type exact numeric, with the decimal precision and scale specified by the <precision> and <scale>.

24) DECIMAL specifies the data type exact numeric, with the decimal scale specified by the <scale> and the implementation-defined decimal precision equal to or greater than the value of the specified <precision>.

25) SMALLINT, INTEGER, and BIGINT specify the data type exact numeric, with scale of 0 (zero) and binary or decimal precision. The choice of binary versus decimal precision is implementation-defined, but the same radix shall be chosen for all three data types. The precision of SMALLINT shall be less than or equal to the precision of INTEGER, and the precision of BIGINT shall be greater than or equal to the precision of INTEGER.

26) FLOAT specifies the data type approximate numeric, with binary precision equal to or greater than the value of the specified <precision>. The maximum value of <precision> is implementation-defined. <precision> shall not be greater than this value.

27) REAL specifies the data type approximate numeric, with implementation-defined precision.

28) DOUBLE PRECISION specifies the data type approximate numeric, with implementation-defined precision that is greater than the implementation-defined precision of REAL.

29) For the <approximate numeric type>s FLOAT, REAL, and DOUBLE PRECISION, the maximum and minimum values of the exponent are implementation-defined.

30) If <time precision> is not specified, then 0 (zero) is implicit. If <timestamp precision> is not specified, then 6 is implicit.

31) If <with or without time zone> is not specified, then WITHOUT TIME ZONE is implicit.

32) The maximum value of <time precision> and the maximum value of <timestamp precision> shall be the same implementation-defined value that is not less than 6. The values of <time precision> and <timestamp precision> shall not be greater than that maximum value.

33) The length of a DATE is 10 positions. The length of a TIME WITHOUT TIME ZONE is 8 positions plus the <time fractional seconds precision>, plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIME WITH TIME ZONE is 14 positions plus the <time fractional seconds precision> plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIMESTAMP WITHOUT TIME ZONE is 19 positions plus the <time fractional seconds precision>, plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIMESTAMP WITH TIME ZONE is 25 positions plus the <time fractional seconds precision> plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero).

34) An <interval type> specifying an <interval qualifier> whose <start field> and <end field> are both either YEAR or MONTH or whose <single datetime field> is YEAR or MONTH is a *year-month interval* type. An <interval type> that is not a year-month interval type is a *day-time interval* type.

NOTE 97 — The length of interval data types is specified in the General Rules of Subclause 10.1, "<interval qualifier>".

35) The *i*-th value of an interval data type corresponds to the *i*-th <primary datetime field>.

36) If <data type> is a <reference type>, then at least one of the following conditions shall be true:

a) There exists a user-defined type descriptor whose user-defined type name is <user-defined type name> *UDTN* simply contained in <referenced type>. *UDTN* shall identify a structured type.

b) <reference type> is contained in the <member list> of <user-defined type definition> *UDTD* and the <path-resolved user-defined type name> simply contained in <referenced type> is equivalent to the <schema-resolved user-defined type name> contained in *UDTD*.

37) The <table name> contained in a <scope clause> shall identify a referenceable table whose structured type is *UDTN*.

38) The <table name> *STN* specified in <scope clause> identifies the scope of the reference type. This scope consists of every row in the table identified by *STN*.

39) An <array type> *AT* specifies an *array type*. The <data type> immediately contained in *AT* is the *element type* of the array type. The <maximum cardinality> immediately contained in *AT* is the *maximum cardinality* of a site of data type *AT*. If the maximum cardinality is not specified, then an implementation-defined maximum cardinality is implicit.

40) A <multiset type> *MT* specifies a *multiset type*. The <data type> immediately contained in *MT* is the *element type* of the multiset type.

41) <row type> specifies the row data type.

42) BOOLEAN specifies the boolean data type.

43) If <data type> *DT1* is contained in a <data type> *DT2*, then the *root data type* of *DT1* is the outermost <data type> that contains *DT1*.

Access Rules

1) If <user-defined type name>, <reference type>, <row type>, or <collection type> *TY* is specified, and *TY* is usage-dependent on a user-defined type *UDT*, then

   Case:

   a) If *TY* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges shall include the USAGE privilege on *UDT*.

   b) Otherwise, the current privileges shall include the USAGE privilege on *UDT*.

General Rules

1) If the result of any specification or operation would be a character value one of whose characters is not in the character set of its declared type, then an exception condition is raised: *data exception — character not in repertoire*.

2) If any specification or operation attempts to cause an item of a character string type whose character set has a character repertoire of UCS to contain a code point that is a noncharacter, then an exception condition is raised: *data exception — noncharacter in UCS string*.

3) If <char length units> other than CHARACTERS is specified, then the conversion of the value of <length> to characters is implementation-defined.

4) For a <datetime type>,

   Case:

a) If DATE is specified, then the data type contains the <primary datetime field>s years, months, and days.

b) If TIME is specified, then the data type contains the <primary datetime field>s hours, minutes, and seconds.

c) If TIMESTAMP is specified, then the data type contains the <primary datetime field>s years, months, days, hours, minutes, and seconds.

d) If WITH TIME ZONE is specified, then the data type contains the time zone datetime fields.

NOTE 98 — Within the non-null values of a <datetime type>, the value of the time zone interval is in the range –14:00 to +14:00. The range for time zone intervals is larger than many readers might expect because it is governed by political decisions in governmental bodies rather than by any natural law.

NOTE 99 — A <datetime type> contains no other fields than those specified by the preceding Rule.

5) For a <datetime type>, a <time fractional seconds precision> that is an explicit or implicit <time precision> or <timestamp precision> defines the number of decimal digits following the decimal point in the SECOND <primary datetime field>.

6) Table 9, "Valid values for datetime fields", specifies the constraints on the values of the <primary datetime field>s in datetime values. The values of TIMEZONE_HOUR and TIMEZONE_MINUTE shall either both be non-negative or both be non-positive.

Table 9 — Valid values for datetime fields

Keyword	Valid values of datetime fields
YEAR	0001 to 9999
MONTH	01 to 12
DAY	Within the range 1 (one) to 31, but further constrained by the value of MONTH and YEAR fields, according to the rules for well-formed dates in the Gregorian calendar.
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 61.9(*N*) where "9(*N*)" indicates the number of digits specified by <time fractional seconds precision>.
TIMEZONE_HOUR	-14 to 14
TIMEZONE_MINUTE	-59 to 59

NOTE 100 — When the value of TIMEZONE_HOUR is either –14 or 14, the value of TIMEZONE_MINUTE is restricted to be 00 (zeros).

NOTE 101 — Datetime data types will allow dates in the Gregorian format to be stored in the date range 0001–01–01 CE through 9999–12–31 CE. The range for SECOND allows for as many as two "leap seconds". Interval arithmetic that involves leap seconds or discontinuities in calendars will produce implementation-defined results.

7) An interval value can be zero, positive, or negative.

8) The values of the <primary datetime field>s within an interval data type are constrained as follows:

   a) The value corresponding to the first <primary datetime field> is an integer with at most *N* digits, where *N* is the <interval leading field precision>.

   b) Table 10, "Valid absolute values for interval fields", specifies the constraints for the absolute values of other <primary datetime field>s in interval values.

   c) If an interval value is zero, then all fields of the interval are zero.

   d) If an interval value is positive, then all fields of the interval are non-negative and at least one field is positive.

   e) If an interval value is negative, then all fields of the interval are non-positive, and at least one field is negative.

Table 10 — Valid absolute values for interval fields

Keyword	Valid values of INTERVAL fields
MONTH	0 to 11
HOUR	0 to 23
MINUTE	0 to 59
SECOND	0 to 59.9(*N*) where "9(*N*)" indicates the number of digits specified by <interval fractional seconds precision> in the <interval qualifier>.

9) If <data type> specifies a character string type, then a character string type descriptor is created, including the following:

   a) The name of the data type (either CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT).

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

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

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

10) If <data type> is a binary string type, then a binary string type descriptor is created, including the following:

   a) The name of the data type (BINARY, BINARY VARYING, or BINARY LARGE OBJECT).

   b) The length or maximum length in octets of the binary string type.

11) If <data type> *DT* specifies an exact numeric type, then:

   a) There shall be an implementation-defined function *ENNF*() that converts any <exact numeric type> *ENT1* into some possibly different <exact numeric type> *ENT2* (the normal form of *ENT1*), subject to the following constraints on *ENNF*():

      i) For every <exact numeric type> *ENT*, *ENNF*(*ENT*) shall not specify DEC or INT.

NOTE 102 — The preceding requirement prohibits the function *ENNF* from returning a value that uses the abbreviated spelling of the two data types; the function shall instead return the long versions of DECIMAL or INTEGER.

ii) For every <exact numeric type> *ENT*, the precision, scale, and radix of *ENNF*(*ENT*) shall be the precision, scale, and radix of *ENT*.

iii) For every <exact numeric type> *ENT*, *ENNF*(*ENT*) shall be the same as *ENNF*(*ENNF*(*ENT*)).

iv) For every <exact numeric type> *ENT*, if *ENNF*(*ENT*) specifies DECIMAL, then *ENNF*(*ENT*) shall specify <precision>, and the precision of *ENNF*(*ENT*) shall be the value of the <precision> specified in *ENNF*(*ENT*).

b) A numeric data type descriptor is created for *DT*, including the following:

i) The name of the type specified in *ENNF*(*DT*) (NUMERIC, DECIMAL, INTEGER, or SMALLINT).

ii) The precision of *DT*.

iii) The scale of *DT*.

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

12) If <data type> *DT* specifies an approximate numeric type, then:

a) There shall be an implementation-defined function *ANNF*() that converts any <approximate numeric type> *ANT* into some possibly different <approximate numeric type> *ANT2* (the normal form of *ANT1*), subject to the following constraints on *ANNF*():

i) For every <approximate numeric type> *ANT*, the precision of *ANNF*(*ANT*) shall be the precision of *ANT*.

ii) For every <approximate numeric type> *ANT*, *ANNF*(*ANT*) shall be the same as *ANNF*(*ANNF*(*ANT*)).

iii) For every <approximate numeric type> *ANT*, if *ANNF*(*ANT*) specifies FLOAT, then *ANNF*(*ANT*) shall specify <precision>, and the precision of *ANNF*(*ANT*) shall be the value of the <precision> specified in *ANNF*(*ANT*).

b) A numeric data type descriptor is created for *DT* including the following:

i) The name of the type specified in *ANNF*(*DT*) (FLOAT, REAL, or DOUBLE PRECISION).

ii) The precision of *DT*.

iii) An indication that the precision is expressed in binary terms.

13) If <data type> specifies <boolean type>, then a boolean data type descriptor is created, including the name of the boolean type (BOOLEAN).

14) If <data type> specifies a <datetime type>, then a datetime data type descriptor is created, including the following:

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

b) The value of the <time fractional seconds precision>, if DATE is not specified.

15) If <data type> specifies an <interval type>, then an interval data type descriptor is created, including the following:

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

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

    c) The <interval qualifier> simply contained in the <interval type>.

16) If <data type> is a <collection type>, then a collection type descriptor is created. Let *KC* be the kind of collection (either ARRAY or MULTISET) specified by <collection type>. Let *ET* be the element type of <collection type>. Let *ETD* be the type designator of *ET*. The collection type descriptor includes the type designator *EDT KC*, an indication of *KC*, the descriptor of *ET*, and (in the case of array types) the maximum cardinality.

17) For a <row type> *RT*, the degree of *RT* is initially set to 0 (zero). The General Rules of Subclause 6.2, "<field definition>", specify the degree of *RT* during the definition of the fields of *RT*.

18) If the <data type> is a <row type>, then a row type descriptor is created. The row type descriptor includes a field descriptor for every <field definition> of the <row type>.

19) A <reference type> identifies a reference type.

20) If <data type> is a <reference type>, then a reference type descriptor is created. Let *RDTN* be the name of the <referenced type>. The reference type descriptor includes the type designator REF(*RDTN*). If a <scope clause> is specified, then the reference type descriptor includes *STN*, identifying the scope of the reference type.

    NOTE 103 — The user-defined type descriptor for a user-defined type is created in the General Rules of Subclause 11.41, "<user-defined type definition>".

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <path-resolved user-defined type name> that identifies a structured type.

2) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean type>.

3) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <time precision> that does not specify 0 (zero).

4) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <timestamp precision> that does not specify either 0 (zero) or 6.

5) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval type>.

6) Without Feature F421, "National character", conforming SQL language shall not contain a <national character string type>

7) Without Feature F411, "Time zone specification", conforming SQL language shall not contain <with or without time zone>.

8) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <reference type>.

9) Without Feature T051, “Row types”, conforming SQL language shall not contain a <row type>.

10) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array type>.

11) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <multiset type>.

12) Without Feature S281, “Nested collection types”, conforming SQL language shall not contain a collection type that is based on a <data type> that contains a <collection type>.

13) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <scope clause> that is not simply contained in a <data type> that is simply contained in a <column definition>.

14) Without Feature S092, “Arrays of user-defined types”, conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

15) Without Feature S272, “Multisets of user-defined types”, conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

16) Without Feature S094, “Arrays of reference types”, conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <reference type>.

17) Without Feature S274, “Multisets of reference types”, conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <reference type>.

18) Without Feature S096, “Optional array bounds”, conforming SQL language shall not contain an <array type> that does not immediately contain <maximum cardinality>.

19) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <binary large object string type>, a <character large object type>, or a <national character large object type>.

20) Without Feature T061, “UCS support”, conforming SQL language shall not contain a <char length units>.

21) Without Feature T071, “BIGINT data type”, conforming SQL language shall not contain BIGINT.

22) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <binary string type> that is not a <binary large object string type>.

6.2 <field definition>

This Subclause is modified by Subclause 6.2, "<field definition>", in ISO/IEC 9075-14.

Function

Define a field of a row type.

Format

```
<field definition> ::=
  <field name> <data type>
```

Syntax Rules

1) Let *RT* be the <row type> that simply contains a <field definition>.

2) The <field name> shall not be equivalent to the <field name> of any other <field definition> simply contained in *RT*.

3) The declared type of the field is <data type>.

4) Let *DT* be the <data type>.

5) If *DT* is CHARACTER or CHARACTER VARYING and does not specify a <character set specification>, then the <character set specification> specified or implicit in the <schema character set specification> is implicit.

Access Rules

None.

General Rules

1) A data type descriptor is created that describes the declared type of the field being defined.

2) The degree of the row type *RT* being defined in the simply containing <row type> is increased by 1 (one).

3) A field descriptor is created that describes the field being defined. The field descriptor includes the following:

   a) The <field name>.

   b) The data type descriptor of the declared type of the field.

   c) The ordinal position of the field in *RT*.

4) The field descriptor is included in the row type descriptor for *RT*.

Conformance Rules

1) Without Feature T051, “Row types”, conforming SQL language shall not contain a <field definition>.

6.3 <value expression primary>

This Subclause is modified by Subclause 6.3, "<value expression primary>", in ISO/IEC 9075-14.

Function

Specify a value that is syntactically self-delimited.

Format

```
<value expression primary> ::=
    <parenthesized value expression>
  | <nonparenthesized value expression primary>

<parenthesized value expression> ::=
  <left paren> <value expression> <right paren>

<nonparenthesized value expression primary> ::=
    <unsigned value specification>
  | <column reference>
  | <set function specification>
  | <window function>
  | <scalar subquery>
  | <case expression>
  | <cast specification>
  | <field reference>
  | <subtype treatment>
  | <method invocation>
  | <static method invocation>
  | <new specification>
  | <attribute or method reference>
  | <reference resolution>
  | <collection value constructor>
  | <array element reference>
  | <multiset element reference>
  | <next value expression>
  | <routine invocation>

<collection value constructor> ::=
    <array value constructor>
  | <multiset value constructor>
```

Syntax Rules

1) The declared type of a <value expression primary> is the declared type of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, or <next value expression>, or the effective returns type of the simply contained <routine invocation>, respectively.

2) Let *NVEP* be a <nonparenthesized value expression primary> of the form "*A.B C*", where *A* satisfies the Format of <schema name>, *B* satisfies the Format of <identifier>, and *C* satisfies the Format of <SQL argument list>. If *NVEP* satisfies the Format, Syntax Rules, and Access Rules of Subclause 6.16, "<method invocation>", then *NVEP* is treated as a <method invocation>; otherwise, *NVEP* is treated as a <routine invocation>.

NOTE 104 — The formal grammar defined in the Format and Syntax Rules of Subclause 6.16, "<method invocation>", and of Subclause 10.4, "<routine invocation>", does not necessarily disambiguate between a <method invocation> and the invocation of a regular function. In such cases, the preceding Syntax Rule ensures that a <nonparenthesized value expression primary> that satisfies the Format, Syntax Rules, and Access Rules of Subclause 6.16, "<method invocation>", is treated as a <method invocation>.

3) The declared type of a <collection value constructor> is the declared type of the <array value constructor> or <multiset value constructor> that it immediately contains.

Access Rules

None.

General Rules

1) The value of a <value expression primary> is the value of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, <next value expression>, or <routine invocation>.

2) The value of a <collection value constructor> is the value of the <array value constructor> or <multiset value constructor> that it immediately contains.

Conformance Rules

None.

6.4 <value specification> and <target specification>

This Subclause is modified by Subclause 6.1, "<value specification> and <target specification>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 6.1, "<value specification> and <target specification>", in ISO/IEC 9075-10.

Function

Specify one or more values, host parameters, SQL parameters, dynamic parameters, or host variables.

Format

```
<value specification> ::=
    <literal>
  | <general value specification>

<unsigned value specification> ::=
    <unsigned literal>
  | <general value specification>

<general value specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <dynamic parameter specification>
  | <embedded variable specification>
  | <current collation specification>
  | CURRENT_CATALOG
  | CURRENT_DEFAULT_TRANSFORM_GROUP
  | CURRENT_PATH
  | CURRENT_ROLE
  | CURRENT_SCHEMA
  | CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>
  | CURRENT_USER
  | SESSION_USER
  | SYSTEM_USER
  | USER
  | VALUE

<simple value specification> ::=
    <literal>
  | <host parameter name>
  | <SQL parameter reference>
  | <embedded variable name>

<target specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <column reference>
  | <target array element specification>
  | <dynamic parameter specification>
  | <embedded variable specification>

<simple target specification> ::=
```

```
    <host parameter specification>
  | <SQL parameter reference>
  | <column reference>
  | <embedded variable name>

<host parameter specification> ::=
  <host parameter name> [ <indicator parameter> ]

<dynamic parameter specification> ::=
  <question mark>

<embedded variable specification> ::=
  <embedded variable name> [ <indicator variable> ]

<indicator variable> ::=
  [ INDICATOR ] <embedded variable name>

<indicator parameter> ::=
  [ INDICATOR ] <host parameter name>

<target array element specification> ::=
  <target array reference>
      <left bracket or trigraph> <simple value specification> <right bracket or trigraph>

<target array reference> ::=
    <SQL parameter reference>
  | <column reference>

<current collation specification> ::=
  COLLATION FOR <left paren> <string value expression> <right paren>
```

Syntax Rules

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

2) Each <host parameter name> shall be contained in an <SQL-client module definition>.

3) If USER is specified, then CURRENT_USER is implicit.

   NOTE 105 — In an environment where the SQL-implementation conforms to Core SQL, conforming SQL language that contains either:

   — A specified or implied <comparison predicate> that compares the <value specification> USER with a <value specification> other than USER, or

   — A specified or implied assignment in which the "value" (as defined in Subclause 9.2, "Store assignment") contains the <value specification> USER

   will become non-conforming in an environment where the SQL-implementation conforms to some SQL package that supports character internationalization, unless the character repertoire of the implementation-defined character set in that environment is identical to the character repertoire of SQL_IDENTIFIER.

4) The declared type of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, and CURRENT_PATH is character string. Whether the character string is fixed length or variable length, and its length if it is fixed length or maximum length if it is variable length, are implementation-defined. The character set of the character string is SQL_IDENTIFIER. The declared type collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

5) The declared type of <string value expression> simply contained in <current collation specification> shall be character string. The declared type of <current collation specification> is character string. Whether the character string is fixed length or variable length, and its length if fixed length or maximum length if variable length, are implementation-defined. The character set of the character string is SQL_IDENTIFIER. The collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

6) The <value specification> or <unsigned value specification> VALUE shall be contained in a <domain constraint>. The declared type of an instance of VALUE is the declared type of the domain to which that domain constraint belongs.

7) A <target specification>, <target array reference>, or <simple target specification> that is a <column reference> shall be a new transition variable column reference.

   NOTE 106 — "new transition variable column reference" is defined in Subclause 6.6, "<identifier chain>".

8) If <target array element specification> is specified, then:

   a) The declared type of the <target array reference> shall be an array type.

   b) The declared type of a <target array element specification> is the element type of the specified <target array reference>.

   c) The declared type of <simple value specification> shall be exact numeric with scale 0 (zero).

9) The declared type of an <indicator variable> shall be exact numeric with a scale of 0 (zero).

10) Each <embedded variable name> shall be contained in an <embedded SQL statement>.

11) Each <dynamic parameter specification> shall be contained in a <preparable statement> that is dynamically prepared in the current SQL-session through the execution of a <prepare statement>.

12) The declared type of CURRENT_DEFAULT_TRANSFORM_GROUP and of CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is a character string. Whether the character string is fixed length or variable length, and its length if fixed length or maximum length if variable length, are implementation-defined. The character set of the character string is SQL_IDENTIFIER. The declared type collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

Access Rules

None.

General Rules

1) A <value specification> or <unsigned value specification> specifies a value that is not selected from a table.

2) A <host parameter specification> identifies a host parameter or a host parameter and an indicator parameter in an <SQL-client module definition>.

3) A <target specification> specifies a target that is a host parameter, an output SQL parameter, a column of a new transition variable, an element of a target whose declared type is an array type, a parameter used in a dynamically prepared statement, or a host variable, according to whether the <target specification> is a <host parameter specification>, an <SQL parameter reference>, a <column reference>, a <target array

element specification>, a <dynamic parameter specification>, or an <embedded variable specification>, respectively.

4) If a <host parameter specification> contains an <indicator parameter> and the value of the indicator parameter is negative, then the value specified by the <host parameter specification> is the null value; otherwise, the value specified by a <host parameter specification> is the value of the host parameter identified by the <host parameter name>.

5) The value specified by a <literal> is the value represented by that <literal>.

6) The value specified by CURRENT_USER is

   Case:

   a) If there is a current user identifier, then the value of that current user identifier.

   b) Otherwise, the null value.

7) The value specified by SESSION_USER is the value of the SQL-session user identifier.

8) The value specified by CURRENT_ROLE is

   Case:

   a) If there is a current role name, then the value of that current role name.

   b) Otherwise, the null value.

9) The value specified by SYSTEM_USER is equal to an implementation-defined string that represents the operating system user who executed the SQL-client module that contains the externally-invoked procedure whose execution caused the SYSTEM_USER <general value specification> to be evaluated.

10) The value specified by CURRENT_CATALOG is the character string that represents the current default catalog name.

11) The value specified by CURRENT_SCHEMA is the character string that represents the current default unqualified schema name.

12) The value specified by CURRENT_PATH is a <schema name list> where <catalog name>s are <delimited identifier>s and the <unqualified schema name>s are <delimited identifier>s. Each <schema name> is separated from the preceding <schema name> by a <comma> with no intervening <space>s. The schemas referenced in this <schema name list> are those referenced in the SQL-path of the current SQL-session context, in the order in which they appear in that SQL-path.

13) The value specified by <current collation specification> is the name of the collation of the <string value expression>.

14) If a <simple value specification> evaluates to the null value, then an exception condition is raised: *data exception — null value not allowed*.

15) A <simple target specification> specifies a target that is a host parameter, an output SQL parameter, a column of a new transition variable, or a host variable, according to whether the <simple target specification> is a <host parameter specification>, an <SQL parameter reference>, a <column reference>, or an <embedded variable name>, respectively.

    NOTE 107 — A <simple target specification> can never be assigned the null value.

16) If a <target specification> or <simple target specification> is assigned a value that is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string*.

17) A <dynamic parameter specification> identifies a parameter used by a dynamically prepared statement.

18) An <embedded variable specification> identifies a host variable or a host variable and an indicator variable.

19) If an <embedded variable specification> contains an <indicator variable> and the value of the indicator variable is negative, then the value specified by the <embedded variable specification> is the null value; otherwise, the value specified by a <embedded variable specification> is the value of the host variable identified by the <embedded variable name>.

20) The value specified by CURRENT_DEFAULT_TRANSFORM_GROUP is the character string that represents the default transform group name in the SQL-session context.

21) The value specified by CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is the character string that represents the transform group name associated with the data type specified by <path-resolved user-defined type name>.

Conformance Rules

1) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_PATH.

2) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <general value specification> that contains VALUE.

3) Without Feature F321, “User authorization”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_USER, SYSTEM_USER, or SESSION_USER.

   NOTE 108 — Although CURRENT_USER and USER are semantically the same, without Feature F321, “User authorization”, CURRENT_USER shall be specified as USER.

4) Without Feature T332, “Extended roles”, conforming SQL language shall not contain CURRENT_ROLE.

5) Without Feature F762, “CURRENT_CATALOG”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_CATALOG.

6) Without Feature F763, “CURRENT_SCHEMA”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_SCHEMA.

7) Without Feature F611, “Indicator data types”, in conforming SQL language, the declared types of <indicator parameter>s and <indicator variable>s shall be the same implementation-defined data type.

8) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <general value specification> that contains a <dynamic parameter specification>.

9) Without Feature S097, “Array element assignment”, conforming SQL language shall not contain a <target array element specification>.

10) Without Feature S241, “Transform functions”, conforming SQL language shall not contain CURRENT_DEFAULT_TRANSFORM_GROUP.

11) Without Feature S241, “Transform functions”, conforming SQL language shall not contain CURRENT_TRANSFORM_GROUP_FOR_TYPE.

12) Without Feature F693, “SQL-session and client module collations”, conforming SQL language shall not contain <current collation specification>.

6.5 <contextually typed value specification>

Function

Specify a value whose data type is to be inferred from its context.

Format

```
<contextually typed value specification> ::=
    <implicitly typed value specification>
  | <default specification>

<implicitly typed value specification> ::=
    <null specification>
  | <empty specification>

<null specification> ::=
  NULL

<empty specification> ::=
    ARRAY <left bracket or trigraph> <right bracket or trigraph>
  | MULTISET <left bracket or trigraph> <right bracket or trigraph>

<default specification> ::=
  DEFAULT
```

Syntax Rules

1) Where the element type *ET* is determined by the context in which *ES* appears, the declared type *DT* of an <empty specification> *ES* is

   Case:

   a) If *ES* simply contains ARRAY, then *ET* ARRAY[0].

   b) If *ES* simply contains MULTISET, then *ET* MULTISET.

   ES is effectively replaced by `CAST ( ES AS DT )`.

   NOTE 109 — In every such context, *ES* is uniquely associated with some expression or site of declared type *DT*, which thereby becomes the declared type of *ES*.

2) The declared type *DT* of a <null specification> *NS* is determined by the context in which *NS* appears. *NS* is effectively replaced by CAST (*NS* AS *DT*).

   NOTE 110 — In every such context, *NS* is uniquely associated with some expression or site of declared type *DT*, which thereby becomes the declared type of *NS*.

3) The declared type *DT* of a <default specification> *DS* is the declared type of a <default option> *DO* included in some site descriptor, determined by the context in which *DS* appears. *DS* is effectively replaced by CAST (*DO* AS *DT*).

   NOTE 111 — In every such context, *DS* is uniquely associated with some site of declared type *DT*, which thereby becomes the declared type of *DS*.

Access Rules

None.

General Rules

1) An <empty specification> specifies a collection whose cardinality is zero.

2) A <null specification> specifies the null value.

3) A <default specification> specifies the default value of some associated item.

Conformance Rules

1) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <empty specification> that simply contains ARRAY.

2) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain an <empty specification> that simply contains MULTISET.

6.6 <identifier chain>

This Subclause is modified by Subclause 6.2, "<identifier chain>", in ISO/IEC 9075-4.

Function

Disambiguate a <period>-separated chain of identifiers.

Format

```
<identifier chain> ::=
  <identifier> [ { <period> <identifier> }... ]

<basic identifier chain> ::=
  <identifier chain>
```

Syntax Rules

1) Let *IC* be an <identifier chain>.

2) Let *N* be the number of <identifier>s immediately contained in *IC*.

3) Let I_i, 1 (one) ≤ i ≤ N, be the <identifier>s immediately contained in *IC*, in order from left to right.

4) Let $PIC_1 = I_1$. For each j between 2 and N, let $PIC_j = PIC_{j-1}$ <period> I_j. PIC_j is called the j-th *partial identifier chain* of *IC*.

5) Let *M* be the minimum of *N* and 4.

6) A column *C* is said to be *refinable* if the declared type of *C* is a row type or a structured type.

7) An SQL parameter *P* is said to be *refinable* if the declared type of *P* is a row type or a structured type.

8) For at most one j between 1 (one) and M, PIC_j is called the *basis* of *IC*, and j is called the *basis length* of *IC*. The *referent* of the basis is a column *C* of a table or an SQL parameter *SP*. The basis, basis length, basis scope, and basis referent of *IC* are determined as follows:

   a) If N = 1 (one), then

      Case:

      i) If *IC* is contained in an <order by clause> of a <cursor specification>, and the <select list> simply contained in the <cursor specification> directly contains a <derived column> *DC* whose explicit or implicit <column name> is equivalent to *IC*, then PIC_1 is a candidate basis, the scope of PIC_1 is the <cursor specification>, and the referent of PIC_1 is the column referenced by *DC*.

      ii) Otherwise, *IC* shall be contained in the scope of one or more range variables whose associated tables include a column whose <column name> is equivalent to I_1 or in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter whose <SQL parameter name> is equivalent to I_1. Let *IS* be the innermost such scope. Let the phrase *possible scope tags* denote those range variables and <routine name>s whose scope is *IS*.

NOTE 112 — "range variable" is defined in Subclause 4.14.6, "Operations involving tables". It is possible for two distinct range variables to be equivalent.

Case:

1) If the number of possible scope is 1 (one), then let *IPST* be that possible scope tag.

   Case:

   A) If *IPST* is a range variable *RV*, then let *T* be the table associated with *RV*. For every column *C* of *T* whose <column name> is equivalent to I_1, PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *RV*, and the referent of PIC_1 is *C*.

      NOTE 113 — Two or more columns with equivalent column names are distinguished by their ordinal positions within *T*.

   B) If *IPST* is a <routine name>, then let *SP* be the SQL parameter whose <SQL parameter name> is equivalent to I_1. PIC_1 is the basis of *IC*, the basis length is 1 (one), the basis scope is the scope of *SP*, and the basis referent is *SP*.

2) Otherwise, each possible scope tag shall be a range variable *RV* of a <table factor> that is directly contained in a <joined table> *JT*. I_1 shall be a common column name in *JT*. Let *C* be the column of *JT* that is identified by I_1. PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *RV*, and the referent of PIC_1 is *C*.

   NOTE 114 — "Common column name" is defined in Subclause 7.7, "<joined table>".

b) If $N > 1$ (one), then the basis, basis length, basis scope, and basis referent are defined in terms of a candidate basis as follows:

   i) If *IC* is contained in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_1, then PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *SP*, and the referent of PIC_1 is *SP*.

   ii) If $N = 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

   iii) If $N > 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes a refinable SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

   iv) If $N = 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. For every column *C* in the table associated with *EN* whose <column name> is equivalent to I_2, PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *EN*, and the referent of PIC_2 is *C*.

   v) If $N > 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. For every refinable column *C* in the table associated with *EN* whose <column name> is equivalent to I_2,

PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *EN*, and the referent of PIC_2 is *C*.

vi) If $N = 2$, 3 or 4, and if PIC_{N-1} is equivalent to an exposed <table or query name> that is in scope, then let *EN* be the exposed <table or query name> that is equivalent to PIC_{N-1} and has the innermost scope. For every column *C* in the table associated with *EN* whose <column name> is equivalent to I_N, PIC_N is a candidate basis of *IC*, the scope of PIC_N is the scope of *EN*, and the referent of PIC_N is *C*.

c) There shall be exactly one candidate basis *CB* with innermost scope. The basis of *IC* is *CB*. The basis length is the length of *CB*. The basis scope is the scope of *CB*. The referent of *IC* is the referent of *CB*.

9) Let *BL* be the basis length of *IC*.

10) If $BL < N$, then let *TIC* be the <value expression primary>:

```
( PIC_BL ) <period>
I_BL+1 <period> ...
<period> I_N
```

The Syntax Rules of Subclause 6.25, "<value expression>", are applied to *TIC*, yielding a column reference or an SQL parameter reference, and $(N - BL)$ <field reference>s or <method invocation>s.

NOTE 115 — In this transformation, (PIC_{BL}) is interpreted as a <value expression primary> of the form <left paren> <value expression> <right paren>. PIC_{BL} is a <value expression> that is a <value expression primary> that is an <unsigned value specification> that is either a <column reference> or an <SQL parameter reference>. The identifiers I_{BL+1}, ..., I_N are parsed using the Syntax Rules of <field reference> and <method invocation>.

11) A <basic identifier chain> shall be an <identifier chain> whose basis is the entire identifier chain.

12) A <basic identifier chain> whose basis referent is a column is a *column reference*. If the basis length is 2, and the basis scope is a <trigger definition> whose <trigger action time> is BEFORE, and I_1 is equivalent to the <new transition variable name> of the <trigger definition>, then the column reference is a *new transition variable column reference*.

13) A <basic identifier chain> whose basis referent is an SQL parameter is an *SQL parameter reference*.

14) The data type of a <basic identifier chain> *BIC* is the data type of the basis referent of *BIC*.

15) If the declared type of a <basic identifier chain> *BIC* is character string, then the collation derivation of the declared type of *BIC* is

Case:

a) If the declared type has a declared type collation *DTC*, then *implicit*.

b) Otherwise, *none*.

Access Rules

None.

General Rules

1) Let *BIC* be a <basic identifier chain>.

2) If *BIC* is a column reference, then *BIC* references the column *C* that is the basis referent of *BIC*.

3) If *BIC* is an SQL parameter reference, then *BIC* references the SQL parameter *SP* of a given invocation of the SQL-invoked routine that contains *SP*.

Conformance Rules

1) Without Feature T325, “Qualified SQL parameter references”, conforming SQL language shall not contain an SQL parameter reference whose first <identifier> is the <qualified identifier> of a <routine name>.

6.7 <column reference>

Function

Reference a column.

Format

```
<column reference> ::=
    <basic identifier chain>
  | MODULE <period> <qualified identifier> <period> <column name>
```

Syntax Rules

1) Every <column reference> has a qualifying table and a qualifying scope, as defined in succeeding Syntax Rules.

2) A <column reference> that is a <basic identifier chain> *BIC* shall be a column reference. The qualifying scope is the basis scope of *BIC* and the qualifying table is the table that contains the basis referent of *BIC*.

3) If MODULE is specified, then <qualified identifier> shall be contained in an <SQL-client module definition> *M*, and shall identify a declared local temporary table *DLTT* whose <temporary table declaration> is contained in *M*, and "MODULE <period> <qualified identifier>" shall be an exposed <table or query name> *MPQI*, and <column name> shall identify a column of *DLTT*. The qualifying table is the table identified by *MPQI*, and the qualifying scope is the scope of *MPQI*.

4) If a <column reference> *CR* is contained in a <table expression> *TE* and the qualifying scope of *CR* contains *TE*, then *CR* is an *outer reference* to the qualifying table of *CR*.

5) Let *C* be the column that is referenced by *CR*. The declared type of *CR* is

   Case:

   a) If the column descriptor of *C* includes a data type,then that data type.

   b) Otherwise, the data type identified in the domain descriptor that describes the domain that is identified by the <domain name> that is included in the column descriptor of *C*.

6) A <column reference> contained in a <query specification> is a *queried column reference*.

7) If *QCR* is a queried column reference, then:

   a) The *qualifying query* of *QCR* is the <query specification> that simply contains the <from clause> that simply contains the <table reference> that defines the qualifying table of *QCR*.

   b) Let *QQ* be the qualifying query of *QCR*.

      Case:

      i) If *QQ* is not grouped, or if *QCR* is contained in the <from clause> or the <where clause> simply contained in *QQ*, then *QCR* is an *ordinary column reference*.

ii) If *QCR* is contained in the <having clause>, <window clause>, or <select list> simply contained in *QQ*, and *QCR* is contained in an aggregated argument of a <set function specification> *SFS*, and *QQ* is the aggregation query of *SFS*, then *QCR* is a *within-group-varying column reference*.

iii) Otherwise, *QCR* is a *group-invariant column reference*.

8) If *QCR* is a group-invariant column reference, then *QCR* shall be functionally dependent on the grouping columns of the qualifying query of *QCR*.

Access Rules

1) Let *CR* be the <column reference>.

2) If the qualifying table of *CR* is a base table or a viewed table, then

Case:

a) If *CR* is contained in a <search condition> immediately contained in an <assertion definition> or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on the column referenced by *CR*.

b) Otherwise,

Case:

i) If *CR* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on the column referenced by *CR*.

ii) Otherwise, the current privileges shall include SELECT on the column referenced by *CR*.

General Rules

1) Let *QCR* be a queried column reference. Let *QT* be the qualifying table of *QCR*, and let *C* be the column of *QT* that is referenced as the basis referent of *QCR*. The value of *QCR* is determined as follows:

a) If *QCR* is an ordinary column reference, then *QCR* denotes the value of *C* in a given row of *QT*.

b) If *QCR* is a within-group-varying column reference, then *QCR* denotes the values of *C* in the rows of a given group of the qualifying query of *QCR* used to construct the argument source of a <set function specification>.

c) If *QCR* is a group-invariant column reference, then *QCR* denotes a value that is not distinct from the value of *C* in every row of a given group of the qualifying query of *QCR*. If the most specific type of *QCR* is character string, datetime with time zone, or user-defined type, then the precise value is chosen in an implementation-dependent fashion.

Conformance Rules

1) Without Feature F821, “Local table references”, conforming SQL language shall not contain a <column reference> that simply contains MODULE.

6.8 <SQL parameter reference>

Function

Reference an SQL parameter.

Format

```
<SQL parameter reference> ::=
  <basic identifier chain>
```

Syntax Rules

1) An <SQL parameter reference> shall be a <basic identifier chain> that is an SQL parameter reference.

2) The declared type of an <SQL parameter reference> is the declared type of the SQL parameter that it references.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

6.9 <set function specification>

Function

Specify a value derived by the application of a function to an argument.

Format

```
<set function specification> ::=
    <aggregate function>
  | <grouping operation>

<grouping operation> ::=
  GROUPING <left paren> <column reference>
      [ { <comma> <column reference> }... ] <right paren>
```

Syntax Rules

1) If <aggregate function> specifies a <general set function>, then the <value expression> simply contained in the <general set function> shall not contain a <set function specification> or a <query expression>.

2) If <aggregate function> specifies <binary set function>, then neither the <dependent variable expression> nor the <independent variable expression> simply contained in the <binary set function> shall contain a <set function specification> or a <query expression>.

3) A <value expression> *VE* simply contained in a <set function specification> *SFE* is an *aggregated argument* of *SFE* if either *SFE* is not an <ordered set function> or *VE* is simply contained in a <within group specification>; otherwise, *VE* is a *non-aggregated argument* of *SFE*.

4) A column reference *CR* contained in an aggregated argument of a <set function specification> *SFS* is called an *aggregated column reference* of *SFS*.

5) If <aggregate function> specifies a <filter clause>, then the <search condition> immediately contained in <filter clause> shall not contain a <set function specification>.

6) The *aggregation query* of a <set function specification> *SFS* is determined as follows.

   Case:

   a) If *SFS* has no aggregated column reference, then the aggregation query of *SFS* is the innermost <query specification> that contains *SFS*.

   b) Otherwise, the innermost qualifying query of the aggregated column references of *SFS* is the aggregation query of *SFS*.

7) *SFS* shall be contained in the <having clause>, <window clause>, or <select list> of its aggregation query.

8) Let *CR* be an aggregated column reference of *SFS* such that the qualifying query *QQ* of *CR* is not the aggregation query of *SFS*. If *QQ* is grouped and *SFS* is contained in the <having clause>, <window clause>, or <select list> of *QQ*, then *CR* shall be functionally dependent on the grouping columns of *QQ*.

9) If <aggregate function> is specified, then the declared type of the result is the declared type of the <aggregate function>.

10) If a <grouping operation> is specified, then:

    a) Each <column reference> shall reference a grouping column of *T*.

    b) The declared type of the result is exact numeric with an implementation-defined precision and a scale of 0 (zero).

    c) If more than one <column reference> is specified, then let *N* be the number of <column reference>s and let CR_i, 1 (one) ≤ *i* ≤ *N*, be the *i*-th <column reference>.

    GROUPING (CR_1, ..., CR_{N-1}, CR_N)

    is equivalent to:

    (2 * GROUPING (CR_1, ..., CR_{N-1}) + GROUPING (CR_N))

Access Rules

None.

General Rules

1) If <aggregate function> is specified, then the result is the value of the <aggregate function>.

NOTE 116 — The value of <grouping operation> is computed by means of syntactic transformations defined in Subclause 7.9, "<group by clause>".

Conformance Rules

1) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <grouping operation>.

2) Without Feature T433, "Multiargument GROUPING function", conforming SQL language shall not contain a <grouping operation> that contains more than one <column reference>.

6.10 <window function>

Function

Specify a window function.

Format

```
<window function> ::=
  <window function type> OVER <window name or specification>

<window function type> ::=
    <rank function type> <left paren> <right paren>
  | ROW_NUMBER <left paren> <right paren>
  | <aggregate function>

<rank function type> ::=
    RANK
  | DENSE_RANK
  | PERCENT_RANK
  | CUME_DIST

<window name or specification> ::=
    <window name>
  | <in-line window specification>

<in-line window specification> ::=
  <window specification>
```

Syntax Rules

1) An <aggregate function> simply contained in a <window function> shall not simply contain a <hypothetical set function>.

2) Let *OF* be the <window function>.

3) Case:

   a) If *OF* is contained in an <order by clause>, then the <order by clause> shall be contained in a <cursor specification> that is a simple table query. Let *ST* be the sort table that is obtained by applying the syntactic transformation of a simple table query, as specified in Subclause 14.1, "<declare cursor>". Let *TE* be the <table expression> contained in the result of that syntactic transformation.

   b) Otherwise, *OF* shall be contained in a <select list> that is immediately contained in a <query specification> *QS* or a <select statement: single row> *SSSR*. Let *QSS* be the innermost <query specification> contained in *QS* that contains *OF*. Let *TE* be the <table expression> immediately contained in *QSS* or *SSSR*.

4) *OF* shall not contain an outer reference or a <query expression>.

5) Let *WNS* be the <window name or specification>. Let *WDX* be a window structure descriptor that describes the window defined by *WNS*.

6) If <rank function type> or ROW_NUMBER is specified, then:

a) If RANK or DENSE_RANK is specified, then the window ordering clause *WOC* of *WDX* shall be present.

b) The window framing of *WDX* shall not be present.

c) Case:

   i) If *WNS* is a <window name>, then let *WNS1* be *WNS*.

   ii) Otherwise, let *WNS1* be the <window specification details> contained in *WNS*.

d) `RANK() OVER` *`WNS`* is equivalent to:

```
( COUNT (*) OVER (WNS1 RANGE UNBOUNDED PRECEDING)
- COUNT (*) OVER (WNS1 RANGE CURRENT ROW) + 1 )
```

e) If DENSE_RANK is specified, then:

   i) Let VE_1, ..., VE_N be an enumeration of the <value expression>s that are <sort key>s simply contained in *WOC*.

   ii) `DENSE_RANK() OVER` *`WNS`* is equivalent to the <window function>:

```
COUNT (DISTINCT ROW ( VE1, ..., VEN ) )
OVER (WNS1 RANGE UNBOUNDED PRECEDING)
```

f) `ROW_NUMBER() OVER` *`WNS`* is equivalent to the <window function>:

```
COUNT (*) OVER (WNS1 ROWS UNBOUNDED PRECEDING)
```

g) Let *ANT1* be an approximate numeric type with implementation-defined precision. `PERCENT_RANK( ) OVER` *`WNS`* is equivalent to:

```
CASE
  WHEN COUNT(*) OVER (WNS1 RANGE BETWEEN UNBOUNDED PRECEDING
                                     AND UNBOUNDED FOLLOWING) = 1
  THEN CAST (0 AS ANT1)
  ELSE
       (CAST (RANK () OVER (WNS1) AS ANT1) - 1) /
       (COUNT (*) OVER (WNS1 RANGE BETWEEN UNBOUNDED PRECEDING
                                AND UNBOUNDED FOLLOWING) - 1)
END
```

h) Let *ANT2* be an approximate numeric type with implementation-defined precision. `CUME_DIST( ) OVER` *`WNS`* is equivalent to:

```
( CAST ( COUNT (*) OVER
     ( WNS1 RANGE UNBOUNDED PRECEDING ) AS ANT2 ) /
   COUNT(*) OVER ( WNS1 RANGE BETWEEN UNBOUNDED PRECEDING
                                  AND UNBOUNDED FOLLOWING ) )
```

7) Let *SL* be the <select list> that simply contains *OF*.

NOTE 117 — If *OF* is originally contained in an <order by clause> of a <cursor specification> that is a simple table query, the syntactic transformation of Subclause 14.3, "<cursor specification>", is to be applied prior to this rule.

8) Let *SQ* be the <set quantifier> of the <query specification> or <select statement: single row> that simply contains *SL*. If there is no <set quantifier>, then let *SQ* be a zero-length string.

9) If <in-line window specification> is specified, then:

   a) Let *WS* be the <window specification>.

   b) Let *WSN* be an implementation-dependent <window name> that is not equivalent to any other <window name> in the <table expression> or <select statement: single row> that simply contains *WS*.

   c) Let *OFT* be the <window function type>.

   d) Let *SLNEW* be the <select list> that is obtained from *SL* by replacing *OF* by:

      ```
      OFT OVER WSN
      ```

   e) Let *FC*, *WC*, *GBC*, and *HC* be <from clause>, <where clause>, <group by clause>, and <having clause>, respectively, of *TE*. If any of <where clause>, <group by clause>, or <having clause> is missing, then let *WC*, *GBC*, or *HC*, respectively, be a zero-length string.

   f) Case:

      i) If there is no <window clause> simply contained in *TE*, then let *WICNEW* be:

         ```
         WINDOW WSN AS WS
         ```

      ii) Otherwise, let *WIC* be the <window clause> simply contained in *TE* and let *WICNEW* be:

         ```
         WIC, WSN AS WS
         ```

   g) Let *TENEW* be:

      ```
      FC WC GBC HC WICNEW
      ```

   h) Case:

      i) If *OF* is simply contained in a <query specification>, then that <query specification> is equivalent to:

         ```
         SELECT SQ SLNEW TENEW
         ```

      ii) Otherwise, *OF* is simply contained in a <select statement: single row>. Let *STL* be the <select target list> of that <select statement: single row>. The <select statement: single row> is equivalent to:

         ```
         SELECT SQ SLNEW INTO STL TENEW
         ```

10) If the window ordering clause or the window framing clause of the window structure descriptor that describes the <window name or specification> is present, then no <aggregate function> simply contained in <window function> shall specify DISTINCT or <ordered set function>.

Access Rules

None.

General Rules

1) The value of <window function> is the value of the <aggregate function>.

Conformance Rules

1) Without Feature T611, “Elementary OLAP operations”, conforming SQL language shall not contain a <window function>.

2) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window name>.

3) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain PERCENT_RANK or CUME_DIST.

4) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window function> that simply contains ROW_NUMBER and immediately contains a <window name or specification> whose window structure descriptor does not contain a window ordering clause.

6.11 <case expression>

This Subclause is modified by Subclause 6.4, "<case expression>", in ISO/IEC 9075-14.

Function

Specify a conditional value.

Format

```
<case expression> ::=
    <case abbreviation>
  | <case specification>

<case abbreviation> ::=
    NULLIF <left paren> <value expression> <comma> <value expression> <right paren>
  | COALESCE <left paren> <value expression>
      { <comma> <value expression> }... <right paren>

<case specification> ::=
    <simple case>
  | <searched case>

<simple case> ::=
  CASE <case operand> <simple when clause>... [ <else clause> ] END

<searched case> ::=
  CASE <searched when clause>... [ <else clause> ] END

<simple when clause> ::=
  WHEN <when operand list> THEN <result>

<searched when clause> ::=
  WHEN <search condition> THEN <result>

<else clause> ::=
  ELSE <result>

<case operand> ::=
    <row value predicand>
  | <overlaps predicate part 1>

<when operand list> ::=
  <when operand> [ { <comma> <when operand> }... ]

<when operand> ::=
    <row value predicand>
  | <comparison predicate part 2>
  | <between predicate part 2>
  | <in predicate part 2>
  | <character like predicate part 2>
  | <octet like predicate part 2>
  | <similar predicate part 2>
  | <regex like predicate part 2>
  | <null predicate part 2>
```

```
  | <quantified comparison predicate part 2>
  | <normalized predicate part 2>
  | <match predicate part 2>
  | <overlaps predicate part 2>
  | <distinct predicate part 2>
  | <member predicate part 2>
  | <submultiset predicate part 2>
  | <set predicate part 2>
  | <type predicate part 2>

<result> ::=
    <result expression>
  | NULL

<result expression> ::=
  <value expression>
```

Syntax Rules

1) If a <case specification> specifies a <case abbreviation>, then:

   a) A <value expression> generally contained in the <case abbreviation> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

   b) `NULLIF (`V_1`, `V_2`)` is equivalent to the following <case specification>:

      ```
      CASE WHEN
      V1=V2 THEN
      NULL ELSE V1
      END
      ```

   c) `COALESCE (`V_1`, `V_2`)` is equivalent to the following <case specification>:

      ```
      CASE WHEN V1
      IS NOT NULL THEN
      V1 ELSE
      V2 END
      ```

   d) `COALESCE (`V_1`, `V_2`, ..., `V_n`)`, for $n \geq 3$, is equivalent to the following <case specification>:

      ```
      CASE WHEN V1
      IS NOT NULL THEN
      V1 ELSE COALESCE
      (V2, ...,
      Vn) END
      ```

2) If a <case specification> specifies a <simple case>, then let *CO* be the <case operand>.

   a) *CO* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

b) If *CO* is <overlaps predicate part 1>, then each <when operand> shall be <overlaps predicate part 2>. If *CO* is <row value predicand>, then no <when operand> shall be an <overlaps predicate part 2>.

c) Let N be the number of <simple when clause>s.

d) For each *i* between 1 (one) and *N*, let WOL_i be the <when operand list> of the *i*-th <simple when clause>. Let $M(i)$ be the number of <when operand>s simply contained in WOL_i. For each *j* between 1 and $M(i)$, let $WO_{i,j}$ be the *j*-th <when operand> simply contained in WOL_i.

e) For each *i* between 1 (one) and *N*, and for each *j* between 1 (one) and $M(i)$,

Case:

i) If $WO_{i,j}$ is a <row value predicand>, then let $EWO_{i,j}$ be

```
= WO_{i,j}
```

ii) Otherwise, let $EWO_{i,j}$ be $WO_{i,j}$.

f) Let R_i be the <result> of the *i*-th <simple when clause>.

g) If <else clause> is specified, then let *CEEC* be that <else clause>; otherwise, let *CEEC* be a character string of length 0 (zero).

h) The <simple case> is equivalent to a <searched case> in which the *i*-th <searched when clause> takes the form:

```
WHEN ( CO EWO_{i,1} ) OR
. . . OR
( CO EWO_{i,M(i)} )
THEN R_i
```

i) The <else clause> of the equivalent <searched case> takes the form:

```
CEEC
```

3) At least one <result> in a <case specification> shall specify a <result expression>.

4) If an <else clause> is not specified, then ELSE NULL is implicit.

5) The declared type of a <case specification> is determined by applying Subclause 9.3, "Result of data type combinations", to the declared types of all <result expression>s in the <case specification>.

Access Rules

None.

General Rules

1) Case:

a) If a <result> specifies NULL, then its value is the null value.

b) If a <result> specifies a <value expression>, then its value is the value of that <value expression>.

2) Case:

a) If the value of the <search condition> of some <searched when clause> in a <case specification> is *True*, then the value of the <case specification> is the value of the <result> of the first (leftmost) <searched when clause> whose <search condition> evaluates to *True*, cast as the declared type of the <case specification>.

b) If no <search condition> in a <case specification> evaluates to *True*, then the value of the <case expression> is the value of the <result> of the explicit or implicit <else clause>, cast as the declared type of the <case specification>.

Conformance Rules

1) Without Feature F262, “Extended CASE expression”, in conforming SQL language, a <case operand> immediately contained in a <simple case> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

2) Without Feature F262, “Extended CASE expression”, in conforming SQL language, a <when operand> contained in a <simple when clause> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

3) Without Feature F263, “Comma-separated predicates in simple CASE expression”, in conforming SQL language, a <when operand list> contained in a <simple when clause> shall simply contain exactly one <when operand>.

6.12 <cast specification>

This Subclause is modified by Subclause 6.2, "<cast specification>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.5, "<cast specification>", in ISO/IEC 9075-14.

Function

Specify a data conversion.

Format

```
<cast specification> ::=
  CAST <left paren> <cast operand> AS <cast target> <right paren>

<cast operand> ::=
    <value expression>
  | <implicitly typed value specification>

<cast target> ::=
    <domain name>
  | <data type>
```

Syntax Rules

1) Case:

   a) If a <domain name> is specified, then let *TD* be the data type of the specified domain.

   b) If a <data type> is specified, then let *TD* be the data type identified by <data type>. <data type> shall not contain a <collate clause>.

2) The declared type of the result of the <cast specification> is *TD*.

3) If the <cast operand> is a <value expression>, then let *SD* be the declared type of the <value expression>.

4) Let *C* be some column and let *CO* be the <cast operand> of a <cast specification> *CS*. *C* is a *leaf column* of *CS* if *CO* consists of a single column reference that identifies *C* or of a single <cast specification> *CS1* of which *C* is a leaf column.

5) If the <cast operand> specifies an <empty specification>, then *TD* shall be a collection type.

6) If the <cast operand> is a <value expression>, then the valid combinations of *TD* and *SD* in a <cast specification> are given by the following table. "Y" indicates that the combination is syntactically valid without restriction; "M" indicates that the combination is valid subject to other Syntax Rules in this Subclause being satisfied; and "N" indicates that the combination is not valid:

SD	*TD*															
	EN	AN	VC	FC	D	T	TS	YM	DT	BO	UDT	CL	B	RT	CT	RW
EN	Y	Y	Y	Y	N	N	N	M	M	N	M	Y	N	M	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N	M	Y	N	M	N	N
C	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	M	Y	N	M	N	N
D	N	N	Y	Y	Y	N	Y	N	N	N	M	Y	N	M	N	N
T	N	N	Y	Y	N	Y	Y	N	N	N	M	Y	N	M	N	N

```
      TS     N   N   Y   Y   Y   Y   Y   N   N   N   M   Y   N   M   N   N
      YM     M   N   Y   Y   N   N   N   Y   N   N   M   Y   N   M   N   N
      DT     M   N   Y   Y   N   N   N   N   Y   N   M   Y   N   M   N   N
      BO     N   N   Y   Y   N   N   N   N   N   Y   M   Y   N   M   N   N
     UDT     M   M   M   M   M   M   M   M   M   M   M   M   M   M   N   N
      B      N   N   N   N   N   N   N   N   N   N   M   N   Y   M   N   N
      RT     M   M   M   M   M   M   M   M   M   M   M   M   M   M   N   N
      CT     N   N   N   N   N   N   N   N   N   N   N   N   N   N   M   N
      RW     N   N   N   N   N   N   N   N   N   N   N   N   N   N   N   M
Where:
      EN  = Exact Numeric
      AN  = Approximate Numeric
      C   = Character (Fixed- or Variable-length, or character large object)
      FC  = Fixed-length Character
      VC  = Variable-length Character
      CL  = Character Large Object
      D   = Date
      T   = Time
      TS  = Timestamp
      YM  = Year-Month Interval
      DT  = Day-Time Interval
      BO  = Boolean
     UDT  = User-Defined Type
      B   = Binary (Fixed- or Variable-Length or Large Object)
      RT  = Reference type
      CT  = Collection type
      RW  = Row type
```

7) If *TD* is an interval and *SD* is exact numeric, then *TD* shall contain only a single <primary datetime field>.

8) If *TD* is exact numeric and *SD* is an interval, then *SD* shall contain only a single <primary datetime field>.

9) If *SD* is character string and *TD* is fixed-length, variable-length, or large object character string, then the character repertoires of *SD* and *TD* shall be the same.

10) If *TD* is a fixed-length, variable-length, or large object character string, then the declared type collation of the <cast specification> is the character set collation of the character set of *TD* and its collation derivation is *implicit*.

11) If the <cast operand> is a <value expression> and either *SD* or *TD* is a user-defined type, then either *TD* shall be a supertype of *SD* or there shall be a data type *P* such that:

    a) The type designator of *P* is in the type precedence list of *SD*.

    b) There is a user-defined cast CF_P whose user-defined cast descriptor includes *P* as the source data type and *TD* as the target data type.

    c) The type designator of no other data type *Q* that is included as the source data type in the user-defined cast descriptor of some user-defined cast CF_Q that has *TD* as the target data type precedes the type designator of *P* in the type precedence list of *SD*.

12) If the <cast operand> is a <value expression> and either *SD* or *TD* is a reference type, then:

    a) Let *RTSD* and *RTTD* be the referenced types of *SD* and *TD*, respectively.

    b) If <data type> is specified and contains a <scope clause>, then let *STD* be that scope. Otherwise, let *STD*, possibly empty, be the scope included in the reference type descriptor of *SD*.

c) Either *RSTD* and *RTTD* shall be compatible, or there shall be a data type *P* in the type precedence list of *SD* such that all of the following are satisfied:

i) There is a user-defined cast CF_P whose user-defined cast descriptor includes *P* as the source data type and *TD* as the target data type.

ii) No other data type *Q* that is included as the source data type in the user-defined cast descriptor of some user-defined cast CF_Q that has *TD* as the target data type precedes *P* in the type precedence list of *SD*.

13) If *SD* is a collection type, then:

a) Let *ESD* be the element type of *SD*.

b) Let *ETD* be the element type of *TD*.

```
CAST ( VALUE AS ETD )
```

where *VALUE* is a <value expression> of declared type *ESD*, shall be a valid <cast specification>.

14) If *SD* is a row type, then:

a) Let *DSD* be the degree of *SD*.

b) Let *DTD* be the degree of *TD*.

c) *DSD* shall be equal to *DTD*.

d) Let FSD_i and FTD_i, 1 (one) ≤ i ≤ *DSD*, be the *i*-th field of *SD* and *TD*, respectively.

e) Let $TFSD_i$ and $TFTD_i$, 1 (one) ≤ i ≤ *DSD*, be the declared type of FSD_i and the declared type of FTD_i, respectively.

f) For *i* varying from 1 (one) to *DSD*, the <cast specification>:

```
CAST ( VALUE_i AS
TFTD_i )
```

where $VALUE_i$ is an arbitrary <value expression> of declared type $TFSD_i$, shall be a valid <cast specification>.

15) If <domain name> is specified, then let *D* be the domain identified by the <domain name>. The schema identified by the explicit or implicit qualifier of the <domain name> shall include the descriptor of *D*.

Access Rules

1) If <domain name> is specified, then

Case:

a) If <cast specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on the domain identified by <domain name>.

b) Otherwise, the current privileges shall include USAGE on the domain identified by <domain name>.

2) If the <cast operand> is a <value expression> and either *SD* or *TD* is a user-defined type, then

Case:

a) If <cast specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include EXECUTE on CF_P.

b) Otherwise, the current privileges shall include EXECUTE on CF_P.

General Rules

1) Let *CS* be the <cast specification>. If the <cast operand> is a <value expression> *VE*, then let *SV* be the value of *VE*.

2) Case:

a) If the <cast operand> specifies NULL, then the result of *CS* is the null value and no further General Rules of this Subclause are applied.

b) If the <cast operand> specifies an <empty specification>, then the result of *CS* is an empty collection of declared type *TD* and no further General Rules of this Subclause are applied.

c) If *SV* is the null value, then the result of *CS* is the null value and no further General Rules of this Subclause are applied.

3) If either *SD* or *TD* is a user-defined type, then

Case:

a) If *TD* is a supertype of *SD*, then *TV* is *SV*.

b) Otherwise:

i) Let *CP* be the cast function contained in the user-defined cast descriptor of CF_P.

ii) Let *SAL* be a static SQL argument list that has a single SQL-argument that is <value expression>. The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SAL* as *STATIC SQL ARG LIST* and *CP* as *SUBJECT ROUTINE*, yielding a value *TR* that is the result of the invocation of *CP*.

iii) Case:

1) If *TD* is a user-defined type, then *TV* is *TR*.

2) Otherwise, *TV* is the result of

```
CAST (TR AS TD)
```

4) If either *SD* or *TD* is a reference type, then

Case:

a) If *RSTD* and *RTTD* are compatible, then:

i) *TV* is *SV*.

ii) The scope in the reference type descriptor of *TV* is *STD*.

b) Otherwise:

i) Let *CP* be the cast function contained in the user-defined cast descriptor of CF_P.

ii) Let *SAL* be a static SQL argument list that has a single SQL-argument that is <value expression>. The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SAL* as *STATIC SQL ARG LIST* and *CP* as *SUBJECT ROUTINE*, yielding value *TV* that is the result of the invocation of *CP*.

iii) The scope in the reference type descriptor of *TV* is *STD*.

5) If *SD* is an array type, then:

a) Let *SC* be the cardinality of *SV*.

b) For *i* varying from 1 (one) to *SC*, the following <cast specification> is applied:

```
CAST ( VE[i] AS ETD )
```

yielding value TVE_i.

c) If *TD* is an array type, then let *TC* be the maximum cardinality of *TD*.

Case:

i) If *SC* is greater than *TC*, then an exception condition is raised: *data exception — array data, right truncation*.

ii) Otherwise, *TV* is the array with elements TVE_i, 1 (one) $\leq i \leq SC$.

d) If *TD* is a multiset type, then *TV* is the multiset with elements TVE_i, 1 (one) $\leq i \leq SC$.

6) If *SD* is a multiset type, then:

a) Let *SC* be the cardinality of *SV*.

b) If *TD* is an array type, then let *TC* be the maximum cardinality of *TD*.

Case:

i) If *SC* is greater than *TC*, then an exception condition is raised: *data exception — array data, right truncation*.

ii) Otherwise, *TV* is the array resulting from the evaluation of

```
ARRAY ( ( SELECT CAST ( M.E AS ETD ) FROM UNNEST (VE) AS M(E) ) )
```

c) If *TD* is a multiset type, then *TV* is the multiset resulting from the evaluation of

```
MULTISET ( ( SELECT CAST ( M.E AS ETD ) FROM UNNEST (VE) AS M(E) ) )
```

7) If *SD* is a row type, then *TV* is the row resulting from the evaluation of

```
ROW ( CAST ( VE.FSD1 AS TFTD1 ),
      CAST ( VE.FSD2 AS TFTD2 ),
   ...
      CAST ( VE.FSDDSD AS TFTDDSD ) )
```

8) If *TD* is exact numeric, then

Case:

a) If *SD* is exact numeric or approximate numeric, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits after rounding or truncating if necessary, then *TV* is that representation. The choice of whether to round or truncate is implementation-defined.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range*.

b) If *SD* is character string, then *SV* is replaced by *SV* with any leading or trailing <space>s removed.

Case:

i) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 5.3, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast*.

ii) Otherwise, let *LT* be that <signed numeric literal>. The <cast specification> is equivalent to

```
CAST ( LT AS TD )
```

c) If *SD* is an interval data type, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits, then *TV* is that representation.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range*.

9) If *TD* is approximate numeric, then

Case:

a) If *SD* is exact numeric or approximate numeric, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits after rounding or truncating if necessary, then *TV* is that representation. The choice of whether to round or truncate is implementation-defined.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range*.

b) If *SD* is character string, then *SV* is replaced by *SV* with any leading or trailing <space>s removed.

Case:

i) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 5.3, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast*.

ii) Otherwise, let *LT* be that <signed numeric literal>. The <cast specification> is equivalent to

```
CAST ( LT AS TD )
```

10) If *TD* is fixed-length character string, then let *LTD* be the length in characters of *TD*.

Case:

a) If *SD* is exact numeric, then:

i) Let *YP* be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 5.3, "<literal>", whose scale is the same as the scale of *SD* and whose interpreted value is the absolute value of *SV*.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast*.

2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD–LY* <space>s.

4) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

b) If *SD* is approximate numeric, then:

i) Let *YP* be a character string as follows.

Case:

1) If *SV* equals 0 (zero), then *YP* is '0E0'.

2) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 5.3, "<literal>", whose interpreted value is equal to the absolute value of *SV* and whose <mantissa> consists of a single <digit> that is not '0' (zero), followed by a <period> and an <unsigned integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast*.

2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD–LY* <space>s.

4) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

c) If *SD* is fixed-length character string, variable-length character string, or large object character string, then

Case:

i) If the length in characters of *SV* is equal to *LTD*, then *TV* is *SV*.

ii) If the length in characters of *SV* is larger than *LTD*, then *TV* is the first *LTD* characters of *SV*. If any of the remaining characters of *SV* are non-<space> characters, then a completion condition is raised: *warning — string data, right truncation*.

iii) If the length in characters *M* of *SV* is smaller than *LTD*, then *TV* is *SV* extended on the right by *LTD–M* <space>s.

d) If *SD* is a datetime data type or an interval data type, then let *Y* be the shortest character string that conforms to the definition of <literal> in Subclause 5.3, "<literal>", and such that the interpreted value of *Y* is *SV* and the interpreted precision of *Y* is the precision of *SD*. If *SV* is an interval, then <sign> shall be specified within <unquoted interval string> in the literal *Y*.

Case:

i) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast*.

ii) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

iii) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD–LY* <space>s.

iv) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

e) If *SD* is boolean, then

Case:

i) If *SV* is *True* and *LTD* is not less than 4, then *TV* is 'TRUE' extended on the right by *LTD*–4 <space>s.

ii) If *SV* is *False* and *LTD* is not less than 5, then *TV* is 'FALSE' extended on the right by *LTD*–5 <space>s.

iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.

11) If *TD* is variable-length character string or large object character string, then let *MLTD* be the maximum length in characters of *TD*.

Case:

a) If *SD* is exact numeric, then:

i) Let *YP* be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 5.3, "<literal>", whose scale is the same as the scale of *SD* and whose interpreted value is the absolute value of *SV*.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-'` || *YP*.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast*.

2) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

b) If *SD* is approximate numeric, then

i) Let *YP* be a character string as follows.

Case:

1) If *SV* equals 0 (zero), then *YP* is '0E0'.

2) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 5.3, "<literal>", whose interpreted value is equal to the absolute value of *SV* and whose <mantissa> consists of a single <digit> that is not '0', followed by a <period> and an <unsigned integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-'` || *YP*.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast*.

2) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

c) If *SD* is fixed-length character string, variable-length character string, or large object character string, then

Case:

i) If the length in characters of *SV* is less than or equal to *MLTD*, then *TV* is *SV*.

ii) If the length in characters of *SV* is larger than *MLTD*, then *TV* is the first *MLTD* characters of *SV*. If any of the remaining characters of *SV* are non-<space> characters, then a completion condition is raised: *warning — string data, right truncation*.

d) If *SD* is a datetime data type or an interval data type then let *Y* be the shortest character string that conforms to the definition of <literal> in Subclause 5.3, "<literal>", and such that the interpreted value of *Y* is *SV* and the interpreted precision of *Y* is the precision of *SD*. If *SV* is a negative interval, then <sign> shall be specified within <unquoted interval string> in the literal *Y*.

Case:

i) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast*.

ii) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

iii) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

e) If *SD* is boolean, then

Case:

i) If *SV* is *True* and *MLTD* is not less than 4, then *TV* is 'TRUE'.

ii) If *SV* is *False* and *MLTD* is not less than 5, then *TV* is 'FALSE'.

iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.

12) If *TD* and *SD* are binary string types, then

Case:

a) If *TD* is fixed-length binary string, then let *LTD* be the length in octets of *TD*.

i) If the length in octets of *SV* is equal to *LTD*, then *TV* is *SV*.

ii) If the length in octets of *SV* is larger than *LTD*, then *TV* is the first *LTD* octets of *SV* and a completion condition is raised: *warning — string data, right truncation*.

iii) If the length in octets *M* of *SV* is smaller than *LTD*, then *TV* is *SV* extended on the right by *LTD–M* X'00's.

b) Otherise, let *MLTD* be the maximum length in octets of *TD*.

Case:

i) If the length in octets of *SV* is less than or equal to *MLTD*, then *TV* is *SV*.

ii) If the length in octets of *SV* is larger than *MLTD*, then *TV* is the first *MLTD* octets of *SV* and a completion condition is raised: *warning — string data, right truncation*.

13) If *TD* is the datetime data type DATE, then

Case:

a) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted date string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) Otherwise, an exception condition is raised: *data exception — invalid datetime format*.

b) If *SD* is the datetime data type DATE, then *TV* is *SV*.

c) If *SD* is the datetime data type TIMESTAMP WITHOUT TIME ZONE, then *TV* is the year, month, and day <primary datetime field>s of *SV*.

d) If *SD* is the datetime data type TIMESTAMP WITH TIME ZONE, then *TV* is computed by:

```
CAST ( CAST ( VE AS TIMESTAMP WITHOUT TIME ZONE ) AS DATE )
```

14) Let *STZD* be the current default time zone displacement of the SQL-session.

15) If *TD* is the datetime data type TIME WITHOUT TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITH TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIME(TSP) WITH TIME ZONE ) AS
TIME(TSP) WITHOUT TIME ZONE )
```

iii) Otherwise, an exception condition is raised: *data exception — invalid datetime format*.

b) If *SD* is TIME WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.

c) If *SD* is TIME WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, computed modulo 24 hours, with implementation-defined rounding or truncation if necessary.

d) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined rounding or truncation if necessary.

e) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIME(TSP) WITHOUT TIME ZONE )
```

16) If *TD* is the datetime data type TIME WITH TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITHOUT TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIME(TSP) WITHOUT TIME ZONE )
AS TIME(TSP) WITH TIME ZONE )
```

iii) Otherwise, an exception condition is raised: *data exception — invalid datetime format*.

b) If *SD* is TIME WITH TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.

c) If *SD* is TIME WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, computed modulo 24 hours, with implementation-defined rounding or truncation if necessary, and the time zone component of *TV* is *STZD*.

d) If *SD* is TIMESTAMP WITH TIME ZONE, then the UTC component of *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined rounding or truncation if necessary, and the time zone component of *TV* is the time zone displacement of *SV*.

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITH TIME ZONE )
AS TIME(TSP) WITH TIME ZONE )
```

17) If *TD* is the datetime data type TIMESTAMP WITHOUT TIME ZONE, then let *TSP* be the <timestamp precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITH TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITH TIME ZONE )
AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
```

iii) Otherwise, an exception condition is raised: *data exception — invalid datetime format*.

b) If *SD* is a date, then the <primary datetime field>s hour, minute, and second of *TV* are set to 0 (zero) and the <primary datetime field>s year, month, and day of *TV* are set to their respective values in *SV*.

c) If *SD* is TIME WITHOUT TIME ZONE, then the <primary datetime field>s year, month, and day of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and second of *TV* are set to their respective values in *SV*, with implementation-defined rounding or truncation if necessary.

d) If *SD* is TIME WITH TIME ZONE, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP WITH TIME ZONE )
AS TIMESTAMP WITHOUT TIME ZONE )
```

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.

f) If *SD* is TIMESTAMP WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, with implementation-defined rounding or truncation if necessary.

18) If *TD* is the datetime data type TIMESTAMP WITH TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then *SV* is replaced by:

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITHOUT TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIMESTAMP(TSP) WITH TIME ZONE )
```

iii) Otherwise, an exception condition is raised: *data exception — invalid datetime format*.

b) If *SD* is a date, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIMESTAMP(TSP) WITH TIME ZONE )
```

c) If *SD* is TIME WITHOUT TIME ZONE, then *TC* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIMESTAMP(TSP) WITH TIME ZONE )
```

d) If *SD* is TIME WITH TIME ZONE, then the <primary datetime field>s of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and

second are set to their respective values in *SV*, with implementation-defined rounding or truncation if necessary. The time zone component of *TV* is set to the time zone component of *SV*.

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, with a time zone displacement of *STZD*.

f) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is *SV* with implementation-defined rounding or truncation, if necessary.

19) If *TD* is interval, then

Case:

a) If *SD* is exact numeric, then

Case:

i) If the representation of *SV* in the data type *TD* would result in the loss of leading significant digits, then an exception condition is raised: *data exception — interval field overflow*.

ii) Otherwise, *TV* is that representation.

b) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted interval string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) Otherwise, an exception condition is raised: *data exception — invalid interval format*.

c) If *SD* is interval and *TD* and *SD* have the same interval precision, then *TV* is *SV*.

d) If *SD* is interval and *TD* and *SD* have different interval precisions, then let *Q* be the least significant <primary datetime field> of *TD*.

i) Let *Y* be the result of converting *SV* to a scalar in units *Q* according to the natural rules for intervals as defined in the Gregorian calendar (that is, there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year).

ii) Normalize *Y* to conform to the <interval qualifier> "*P* TO *Q*" of *TD* (again, observing the rules that there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year). Whether to truncate or round in the least significant field of the result is implementation-defined. If this would result in loss of precision of the leading datetime field of *Y*, then an exception condition is raised: *data exception — interval field overflow*.

iii) *TV* is the value of *Y*.

20) If *TD* is boolean, then

Case:

a) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.

b) If *SD* is boolean, then *TV* is *SV*.

21) If the <cast specification> contains a <domain name> and that <domain name> refers to a domain that contains a <domain constraint> and if *TV* does not satisfy the <check constraint definition> simply contained in the <domain constraint>, then an exception condition is raised: *integrity constraint violation*.

22) The result of *CS* is *TV*.

Conformance Rules

1) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain a <cast operand> whose declared type is BINARY LARGE OBJECT or CHARACTER LARGE OBJECT.

2) Without Feature F421, "National character", conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER LARGE OBJECT.

3) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER LARGE OBJECT.

4) Without Feature S043, "Enhanced reference types", in conforming SQL language, if the declared data type of <cast operand> is a reference type, then <cast target> shall contain a <data type> that is a reference type.

6.13 <next value expression>

This Subclause is modified by Subclause 6.3, "<next value expression>", in ISO/IEC 9075-4.

Function

Return the next value of a sequence generator.

Format

```
<next value expression> ::=
  NEXT VALUE FOR <sequence generator name>
```

Syntax Rules

1) A <next value expression> shall be directly contained in one of the following:

   a) A <select list> simply contained in a <query specification> that constitutes a <query expression> that is immediately contained in one of the following:

      i) A <cursor specification>.

      ii) A <subquery> simply contained in an <as subquery clause> in a <table definition>.

      iii) A <from subquery>.

      iv) A <select statement: single row>.

   b) A <select list> simply contained in a <query specification> that is immediately contained in a <dynamic single row select statement>.

   c) A <from constructor>.

   d) A <merge insert value list>.

   e) An <update source>.

2) <next value expression> shall not be contained in a <case expression>, a <search condition>, an <order by clause>, an <aggregate function>, a <window function>, a grouped query, or in a <query specification> that simply contains the <set quantifier> DISTINCT.

3) The declared type of <next value expression> is the data type described by the data type descriptor included in the sequence generator descriptor identified by <sequence generator name>.

Access Rules

1) Case:

   a) If <next value expression> is contained in a <schema definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include USAGE privilege on the sequence generator identified by <sequence generator name>.

b) Otherwise, the current privileges shall include USAGE privilege on the sequence generator identified by <sequence generator name>.

General Rules

1) If <next value expression> *NVE* is specified, then let *SEQ* be the sequence generator descriptor identified by the <sequence generator name> contained in *NVE*.

Case:

a) If *NVE* is directly contained in a <query specification> *QS*, then the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator", are applied once per row in the result of *QS* with *SEQ* as *SEQUENCE*. The result of each evaluation of *NVE* for a given row is the *RESULT* returned by the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator".

b) If *NVE* is directly contained in a <contextually typed table value constructor> *TVC*, then the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator", are applied once per <contextually typed row value expression> contained in *TVC*. The result of each evaluation of *NVE* for a given <row value expression> is the *RESULT* returned by the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator".

c) If *NVE* is directly contained in an <update source>, then the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator", are applied once per row to be updated by the <update statement: searched> or <update statement: positioned>. The result of each evaluation of *NVE* for a given row is the *RESULT* returned by the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator".

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <next value expression>.

6.14 <field reference>

Function

Reference a field of a row value.

Format

```
<field reference> ::=
  <value expression primary> <period> <field name>
```

Syntax Rules

1) Let *FR* be the <field reference>, let *VEP* be the <value expression primary> immediately contained in *FR*, and let *FN* be the <field name> immediately contained in *FR*.

2) The declared type of *VEP* shall be a row type. Let *RT* be that row type.

3) *FR* is a *field reference*.

4) *FN* shall unambiguously reference a field of *RT*. Let *F* be that field.

5) The declared type of *FR* is the declared type of *F*.

Access Rules

None.

General Rules

1) Let *VR* be the value of *VEP*.

2) Case:

   a) If *VR* is the null value, then the value of *FR* is the null value.

   b) Otherwise, the value of *FR* is the value of the field *F* of *VR*.

Conformance Rules

1) Without Feature T051, “Row types”, conforming SQL language shall not contain a <field reference>.

6.15 <subtype treatment>

Function

Modify the declared type of an expression.

Format

```
<subtype treatment> ::=
  TREAT <left paren> <subtype operand> AS <target subtype> <right paren>

<subtype operand> ::=
  <value expression>

<target subtype> ::=
    <path-resolved user-defined type name>
  | <reference type>
```

Syntax Rules

1) The declared type *VT* of the <value expression> shall be a structured type or a reference type.

2) Case:

   a) If *VT* is a structured type, then:

      i) <target subtype> shall specify a <path-resolved user-defined type name>.

      ii) Let *DT* be the structured type identified by the <user-defined type name> simply contained in <path-resolved user-defined type name>.

   b) Otherwise:

      i) <target subtype> shall specify a <reference type>.

      ii) Let *DT* be the reference type identified by <reference type>.

3) *VT* shall be a supertype of *DT*.

4) The declared type of the result of the <subtype treatment> is *DT*.

Access Rules

None.

General Rules

1) Let *V* be the value of the <value expression>.

2) Case:

   a) If *V* is the null value, then the value of the <subtype treatment> is the null value.

b) Otherwise:

i) If the most specific type of *V* is not a subtype of *DT*, then an exception condition is raised: *invalid target type specification*.

NOTE 118 — "most specific type" is defined in Subclause 4.7.5, "Subtypes and supertypes".

ii) The value of the <subtype treatment> is *V*.

Conformance Rules

1) Without Feature S161, "Subtype treatment", conforming SQL Language shall not contain a <subtype treatment>.

2) Without Feature S162, "Subtype treatment for references", conforming SQL language shall not contain a <target subtype> that contains a <reference type>.

6.16 <method invocation>

This Subclause is modified by Subclause 6.1, "<method invocation>", in ISO/IEC 9075-13.

Function

Reference an SQL-invoked method of a user-defined type value.

Format

```
<method invocation> ::=
    <direct invocation>
  | <generalized invocation>

<direct invocation> ::=
  <value expression primary> <period> <method name> [ <SQL argument list> ]

<generalized invocation> ::=
  <left paren> <value expression primary> AS <data type> <right paren>
      <period> <method name> [ <SQL argument list> ]

<method selection> ::=
  <routine invocation>

<constructor method selection> ::=
  <routine invocation>
```

Syntax Rules

1) Let *OR* be the <method invocation>, let *VEP* be the <value expression primary> immediately contained in the <direct invocation> or <generalized invocation> of *OR*, and let *MN* be the <method name> immediately contained in *OR*.

2) The declared type of *VEP* shall be a user-defined type. Let *UDT* be that user-defined type.

3) Case:

   a) If <SQL argument list> is specified, then let *AL* be:

      $, A_1, ..., A_n$

      where A_i, 1 (one) ≤ i ≤ n, are the <SQL argument>s immediately contained in <SQL argument list>, taken in order of their ordinal position in <SQL argument list>.

   b) Otherwise, let *AL* be a zero-length string.

4) Case:

   a) If <method invocation> is immediately contained in <new invocation>, then let *TP* be an SQL-path containing the <schema name> of the schema that includes the descriptor of *UDT*.

   b) Otherwise, let *TP* be an SQL-path, arbitrarily defined, containing the <schema name> of every schema that includes a descriptor of a supertype or subtype of *UDT*.

5) Case:

   a) If <generalized invocation> is specified, then let *DT* be the <data type> simply contained in the <generalized invocation>. Let *RI* be the following <method selection>:

      ```
      MN (VEP AS DT AL)
      ```

   b) Otherwise,

      Case:

      i) If <method invocation> is immediately contained in <new invocation>, then let *RI* be the <constructor method selection>:

         ```
         MN (VEP AL)
         ```

      ii) Otherwise, let *RI* be the following <method selection>:

         ```
         MN (VEP AL)
         ```

6) The Syntax Rules of Subclause 10.4, "<routine invocation>", are applied with *RI* as *ROUTINE INVOCATION* and *TP* as *SQLPATH*, yielding subject routine *SR* and static SQL argument list *SAL*.

Access Rules

None.

General Rules

1) The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*, yielding value *V* that is the result of the <routine invocation>.

2) The value of <method invocation> is *V*.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <method invocation>.

6.17 <static method invocation>

Function

Invoke a static method.

Format

```
<static method invocation> ::=
  <path-resolved user-defined type name> <double colon> <method name>
      [ <SQL argument list> ]

<static method selection> ::=
  <routine invocation>
```

Syntax Rules

1) Let *TN* be the <user-defined type name> immediately contained in <path-resolved user-defined type name> and let *T* be the user-defined type identified by *TN*.

2) Let *MN* be the <method name> immediately contained in <static method invocation>.

3) Case:

   a) If <SQL argument list> is specified, then let *AL* be that <SQL argument list>.

   b) Otherwise, let *AL* be <left paren> <right paren>.

4) Let *TP* be an SQL-path containing only the <schema name> of every schema that includes a descriptor of a supertype of *T*.

5) Let *RI* be the following <routine invocation>:

   ```
   MN AL
   ```

6) Let *SMS* be the following <static method selection>:

   ```
   RI
   ```

7) The Syntax Rules of Subclause 10.4, "<routine invocation>", are applied with *RI* as *ROUTINE INVOCATION*, *TP* as *SQLPATH*, and *T* as *UDT*, yielding subject routine *SR* and static SQL argument list *SAL*.

Access Rules

None.

General Rules

1) The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*, yielding a value *V* that is the result of the <routine invocation>.

2) The value of <static method invocation> is *V*.

Conformance Rules

1) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <static method invocation>.

6.18 <new specification>

This Subclause is modified by Subclause 6.2, "<new specification>", in ISO/IEC 9075-13.

Function

Invoke a method on a newly-constructed value of a structured type.

Format

```
<new specification> ::=
  NEW <path-resolved user-defined type name> <SQL argument list>

<new invocation> ::=
    <method invocation>
  | <routine invocation>
```

Syntax Rules

1) Let *UDTN* be the <path-resolved user-defined type name> immediately contained in the <new specification>. Let *MN* be the <qualified identifier> immediately contained in *UDTN*.

2) Let *UDT* be the user-defined type identified by *UDTN*. *UDT* shall be instantiable. Let *SN* be the implicit or explicit <schema name> of *UDTN*. Let *S* be the schema identified by *SN*. Let *RN* be *NS.MN*.

3) Case:

   a) If the <new specification> is of the form

      NEW *UDTN*()

      then

      Case:

      i) If *S* does not include the descriptor of an SQL-invoked constructor method whose method name is equivalent to *MN* and whose unaugmented parameter list is empty, then the <new specification> is equivalent to the <new invocation>

         RN()

      ii) Otherwise, the <new specification> is equivalent to the <new invocation>

         RN().*MN*()

   b) Otherwise, the <new specification>

      NEW *UDTN*(a_1, a_2, ..., a_n)

      is equivalent to the <new invocation>

      RN().*MN*(a_1, a_2, ..., a_n)

Access Rules

None.

NOTE 119 — The applicable privileges or current privileges (as appropriate) include EXECUTE privilege on the constructor function, and also on the indicated constructor method, according to the Syntax Rules of Subclause 10.4, "<routine invocation>".

General Rules

None.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <new specification>.

6.19 <attribute or method reference>

Function

Return a value acquired by accessing a column of the row identified by a value of a reference type or by invoking an SQL-invoked method.

Format

```
<attribute or method reference> ::=
  <value expression primary> <dereference operator> <qualified identifier>
      [ <SQL argument list> ]

<dereference operator> ::=
  <right arrow>
```

Syntax Rules

1) The declared type of the <value expression primary> *VEP* shall be a reference type and the scope included in its reference type descriptor shall not be empty. Let *RT* be the referenced type of *VEP*.

2) Let *QI* be the <qualified identifier>. If <SQL argument list> is specified, then let *SAL* be <SQL argument list>; otherwise, let *SAL* be a zero-length string.

3) Case:

   a) If *QI* is equivalent to the attribute name of an attribute of *RT* and *SAL* is a zero-length string, then <attribute or method reference> is effectively replaced by a <dereference operation> *AMR* of the form:

      ```
      VEP -> QI
      ```

   b) Otherwise, <attribute or method reference> is effectively replaced by a <method reference> *AMR* of the form:

      ```
      VEP -> QI SAL
      ```

4) The declared type of <attribute or method reference> is the declared type of *AMR*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain an <attribute or method reference>.

6.20 <dereference operation>

Function

Access a column of the row identified by a value of a reference type.

Format

```
<dereference operation> ::=
  <reference value expression> <dereference operator> <attribute name>
```

Syntax Rules

1) Let *RVE* be the <reference value expression>. The reference type descriptor of *RVE* shall include a scope. Let *RT* be the referenced type of *RVE*.

2) Let *AN* be the <attribute name>. *AN* shall identify an attribute *AT* of *RT*.

3) The declared type of the <dereference operation> is the declared type of *AT*.

4) Let *S* be the name of the referenceable table in the scope of the reference type of *RVE*.

5) Let *OID* be the name of the self-referencing column of *S*.

6) <dereference operation> is equivalent to a <scalar subquery> of the form:

```
( SELECT AN
  FROM S
  WHERE S.OID = RVE )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <dereference operation>.

6.21 <method reference>

Function

Return a value acquired from invoking an SQL-invoked routine that is a method.

Format

```
<method reference> ::=
  <value expression primary> <dereference operator> <method name> <SQL argument list>
```

Syntax Rules

1) The data type of the <value expression primary> *VEP* shall be a reference type and the scope included in its reference type descriptor shall not be empty.

2) Let *MN* be the method name. Let *MRAL* be the <SQL argument list>.

3) The Syntax Rules of Subclause 6.16, "<method invocation>", are evaluated for the <method invocation>:

   `DEREF (`*`VEP`*`) . `*`MN MRAL`

   yielding subject routine *SR* and static SQL argument list *SAL*.

4) The data type of <method reference> is the data type of the expression:

   `DEREF (`*`VEP`*`) . `*`MN MRAL`

Access Rules

1) Let *SCOPE* be the table that is the scope of *VEP*.

   Case:

   a) If <method reference> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include the table/method privilege for table *SCOPE* and method *SR*.

   b) Otherwise, the current privileges shall include the table/method privilege for table *SCOPE* and method *SR*.

General Rules

1) The General Rules of Subclause 6.16, "<method invocation>", are applied with *SR* and *SAL* as the subject routine and SQL argument list, respectively, yielding a value *V* that is the result of the <routine invocation>.

2) The value of <method reference> is *V*.

Conformance Rules

1) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <method reference>.

6.22 <reference resolution>

Function

Obtain the value referenced by a reference value.

Format

```
<reference resolution> ::=
  DEREF <left paren> <reference value expression> <right paren>
```

Syntax Rules

1) Let *RR* be the <reference resolution> and let *RVE* be the <reference value expression>. The reference type descriptor of *RVE* shall include a scope.

2) The declared type of *RR* is the structured type that is referenced by the declared type of *RVE*.

3) Let *SCOPE* be the table identified by the table name included in the reference type descriptor of *RVE*. *SCOPE* is the scoped table of *RR*.

   NOTE 120 — The term "scoped table" is defined in Subclause 4.9, "Reference types".

4) Let m be the number of subtables of *SCOPE*. Let S_i, 1 (one) $\le i \le m$, be the subtables, arbitrarily ordered, of *SCOPE*.

5) For each S_i, 1 (one) $\le i \le m$, let STN_i be the name included in the descriptor of S_i of the structured type ST_i associated with S_i, let $REFCOL_i$ be the self-referencing column of S_i, let N_i be the number of attributes of ST_i, and let $A_{i,j}$, 1 (one) $\le j \le N_i$, be the names of the attributes of ST_i, therefore also the names of the columns of S_i.

Access Rules

1) Case:

   a) If <reference resolution> is contained in a <schema definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *SCOPE*.

   b) Otherwise, the current privileges shall includes SELECT WITH HIERARCHY OPTION on at least one supertable of *SCOPE*.

General Rules

1) The value of <reference resolution> is the value of:

   (SELECT $A_{1,1}$ (... A_{1,N_1}

```
        ( STN1(), A1,N1 ), ... A1,1 )
  FROM ONLY S1
  WHERE S1.REFCOL1 = RVE
 UNION
  SELECT A2,1 ( ... A2,N2
        ( STN2(), A2,N2 ), ... A2,1 )
  FROM ONLY S2
  WHERE S2.REFCOL2 = RVE
 UNION
 ...
 UNION
  SELECT Am,1 ( ... Am,Nm
        ( STNm(), Am,Nm ), ... Am,1 )
  FROM ONLY Sm
  WHERE Sm.REFCOLm = RVE
)
```

NOTE 121 — The evaluation of this General Rule is effectively performed without further Access Rule checking.

Conformance Rules

1) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <reference resolution>.

6.23 <array element reference>

Function

Return an element of an array.

Format

```
<array element reference> ::=
  <array value expression>
      <left bracket or trigraph> <numeric value expression> <right bracket or trigraph>
```

Syntax Rules

1) The declared type of an <array element reference> is the element type of the specified <array value expression>.

2) The declared type of <numeric value expression> shall be exact numeric with scale 0 (zero).

Access Rules

None.

General Rules

1) If the value of <array value expression> or <numeric value expression> is the null value, then the result of <array element reference> is the null value.

2) Let the value of <numeric value expression> be *i*.

   Case:

   a) If *i* is greater than zero and less than or equal to the cardinality of <array value expression>, then the result of <array element reference> is the value of the *i*-th element of the value of <array value expression>.

   b) Otherwise, an exception condition is raised: *data exception — array element error*.

Conformance Rules

1) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array element reference>.

6.24 <multiset element reference>

Function

Return the sole element of a multiset of one element.

Format

```
<multiset element reference> ::=
  ELEMENT <left paren> <multiset value expression> <right paren>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression>. The <multiset element reference> is equivalent to the <scalar subquery>

```
( SELECT M.E
  FROM UNNEST (MVE) AS M(E) )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <multiset element reference>.

6.25 <value expression>

This Subclause is modified by Subclause 6.3, "<value expression>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.7, "<value expression>", in ISO/IEC 9075-14.

Function

Specify a value.

Format

```
<value expression> ::=
    <common value expression>
  | <boolean value expression>
  | <row value expression>

<common value expression> ::=
    <numeric value expression>
  | <string value expression>
  | <datetime value expression>
  | <interval value expression>
  | <user-defined type value expression>
  | <reference value expression>
  | <collection value expression>

<user-defined type value expression> ::=
  <value expression primary>

<reference value expression> ::=
  <value expression primary>

<collection value expression> ::=
    <array value expression>
  | <multiset value expression>
```

Syntax Rules

1) The declared type of a <value expression> is the declared type of the simply contained <common value expression>, <boolean value expression>, or <row value expression>.

2) The declared type of a <common value expression> is the declared type of the <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <user-defined type value expression>, <collection value expression>, or <reference value expression>, respectively.

3) The declared type of a <user-defined type value expression> is the declared type of the immediately contained <value expression primary>, which shall be a user-defined type.

4) The declared type of a <reference value expression> is the declared type of the immediately contained <value expression primary>, which shall be a reference type.

5) The declared type of a <collection value expression> is the declared type of the immediately contained <array value expression> or <multiset value expression>.

6) Let *C* be some column. Let *VE* be the <value expression>. *C* is an underlying column of *VE* if and only if *C* is identified by some column reference contained in *VE*.

7) A <value expression> or <nonparenthesized value expression primary> is *possibly non-deterministic* if it generally contains any of the following:

   a) A <datetime value function>.

   b) A <next value expression>.

   c) A <cast specification> that either is, or recursively implies through the execution of the General Rules of Subclause 6.12, "<cast specification>", one of the following:

      i) A <cast specification> whose result type is datetime with time zone and whose <cast operand> has declared type that is not datetime with time zone.

      ii) A <cast specification> whose result type is an array type and whose <cast operand> has a declared type that is a multiset type.

   d) An <array value constructor by query>.

   e) A <datetime factor> that simply contains a <datetime primary> whose declared type is datetime without time zone and that simply contains an explicit <time zone>.

   f) An <interval value expression> that computes the difference of a <datetime value expression> and a <datetime term>, such that the declared type of one operand is datetime with time zone and the other operand is datetime without time zone.

   g) A <comparison predicate> , <overlaps predicate>, or <distinct predicate> simply containing <row value predicand>s *RVP1* and *RVP2* such that the declared types of *RVP1* and *RVP2* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

      NOTE 122 — This includes <between predicate> because of a syntactic transformation to <comparison predicate>.

   h) A <quantified comparison predicate> or a <match predicate> simply containing a <row value predicand> *RVP* and a <table subquery> *TS* such that the declared types of *RVP* and *TS* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

      NOTE 123 — This includes <in predicate> because of a syntactic transformation to <quantified comparison predicate>.

   i) A <member predicate> simply containing a <row value predicand> *RVP* and a <multiset value expression> *MVP* such that the declared type of the only field *F* of *RVP* and the element type of *MVP* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

   j) A <submultiset predicate> simply containing a <row value predicand> *RVP* and a <multiset value expression> *MVP* such that the declared type of the only field *F* of *RVP* and the declared type of *MVP* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

   k) A <multiset value expression> that specifies or implies MULTISET UNION, MULTISET EXCEPT, or MULTISET INTERSECT such that the element types of the operands have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

   l) A <value specification> that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH.

m) A <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.

n) An <aggregate function> that specifies MIN or MAX and that simply contains a <value expression> whose declared type is based on a character string type, user-defined type, or datetime with time zone type.

o) An <aggregate function> that specifies INTERSECTION and that simply contains a <value expression> whose declared element type is based on a character string type, a user-defined type, or a datetime type with time zone.

p) A <multiset value expression> that specifies MULTISET UNION DISTINCT, MULTISET EXCEPT, or MULTISET INTERSECT and whose result type's declared element type is based on character string type, a user-defined type, or a datetime type with time zone.

q) A <multiset set function> whose declared element type is based on a character string type, a user-defined type, or a datetime type with time zone.

r) A <window function> that specifies ROW_NUMBER or whose associated <window specification> specifies ROWS.

s) A <query specification> or <query expression> that is possibly non-deterministic.

NOTE 124 — The notion of one data type being based on another data type is defined in Subclause 4.1, “Data types”.

Access Rules

None.

General Rules

1) The value of a <value expression> is the value of the simply contained <common value expression>, <boolean value expression>, or <row value expression>.

2) The value of a <common value expression> is the value of the immediately contained <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <user-defined type value expression>, <collection value expression>, or <reference value expression>.

3) When a <value expression> *V* is evaluated for a row *R* of a table, each reference to a column of that table by a column reference *CR* directly contained in *V* is the value of that column in that row.

4) The value of a <collection value expression> is the value of its immediately contained <array value expression> or <multiset value expression>.

5) The value of a <reference value expression> *RVE* is the value of the <value expression primary> immediately contained in *RVE*.

Conformance Rules

1) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <value expression> that is a <boolean value expression>.

2) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <reference value expression>.

6.26 <numeric value expression>

Function

Specify a numeric value.

Format

```
<numeric value expression> ::=
    <term>
  | <numeric value expression> <plus sign> <term>
  | <numeric value expression> <minus sign> <term>

<term> ::=
    <factor>
  | <term> <asterisk> <factor>
  | <term> <solidus> <factor>

<factor> ::=
  [ <sign> ] <numeric primary>

<numeric primary> ::=
    <value expression primary>
  | <numeric value function>
```

Syntax Rules

1) If the declared type of both operands of a dyadic arithmetic operator is exact numeric, then the declared type of the result is an implementation-defined exact numeric type, with precision and scale determined as follows:

   a) Let *S1* and *S2* be the scale of the first and second operands respectively.

   b) The precision of the result of addition and subtraction is implementation-defined, and the scale is the maximum of *S1* and *S2*.

   c) The precision of the result of multiplication is implementation-defined, and the scale is *S1* + *S2*.

   d) The precision and scale of the result of division are implementation-defined.

2) If the declared type of either operand of a dyadic arithmetic operator is approximate numeric, then the declared type of the result is an implementation-defined approximate numeric type.

3) The declared type of a <factor> is that of the immediately contained <numeric primary>.

4) The declared type of a <numeric primary> shall be numeric.

5) If a <numeric value expression> immediately contains a <minus sign> *NMS* and immediately contains a <term> that immediately contains a <factor> that immediately contains a <sign> that is a <minus sign> *FMS*, then there shall be a <separator> between *NMS* and *FMS*.

Access Rules

None.

General Rules

1) If the value of any <numeric primary> simply contained in a <numeric value expression> is the null value, then the result of the <numeric value expression> is the null value.

2) If the <numeric value expression> contains only a <numeric primary>, then the result of the <numeric value expression> is the value of the specified <numeric primary>.

3) The monadic arithmetic operators <plus sign> and <minus sign> (+ and –, respectively) specify monadic plus and monadic minus, respectively. Monadic plus does not change its operand. Monadic minus reverses the sign of its operand.

4) The dyadic arithmetic operators <plus sign>, <minus sign>, <asterisk>, and <solidus> (+, –, *, and /, respectively) specify addition, subtraction, multiplication, and division, respectively. If the value of a divisor is zero, then an exception condition is raised: *data exception — division by zero*.

5) If the most specific type of the result of an arithmetic operation is exact numeric, then

   Case:

   a) If the operator is not division and the mathematical result of the operation is not exactly representable with the precision and scale of the result data type, then an exception condition is raised: *data exception — numeric value out of range*.

   b) If the operator is division and the approximate mathematical result of the operation represented with the precision and scale of the result data type loses one or more leading significant digits after rounding or truncating if necessary, then an exception condition is raised: *data exception — numeric value out of range*. The choice of whether to round or truncate is implementation-defined.

6) If the most specific type of the result of an arithmetic operation is approximate numeric and the exponent of the approximate mathematical result of the operation is not within the implementation-defined exponent range for the result data type, then an exception condition is raised: *data exception — numeric value out of range*.

Conformance Rules

None.

6.27 <numeric value function>

Function

Specify a function yielding a value of type numeric.

Format

```
<numeric value function> ::=
    <position expression>
  | <regex occurrences function>
  | <regex position expression>
  | <extract expression>
  | <length expression>
  | <cardinality expression>
  | <absolute value expression>
  | <modulus expression>
  | <natural logarithm>
  | <exponential function>
  | <power function>
  | <square root>
  | <floor function>
  | <ceiling function>
  | <width bucket function>

<position expression> ::=
    <character position expression>
  | <binary position expression>

<regex occurrences function> ::=
  OCCURRENCES_REGEX <left paren>
      <XQuery pattern> [ FLAG <XQuery option flag> ]
      IN <regex subject string>
      [ FROM <start position> ]
      [ USING <char length units> ]
      <right paren>

<XQuery pattern> ::=
  <character value expression>

<XQuery option flag> ::=
  <character value expression>

<regex subject string> ::=
  <character value expression>

<regex position expression> ::=
  POSITION_REGEX <left paren>
      [ <regex position start or after> ]
      <XQuery pattern> [ FLAG <XQuery option flag> ]
      IN <regex subject string>
      [ FROM <start position> ]
      [ USING <char length units> ]
      [ OCCURRENCE <regex occurrence> ]
      [ GROUP <regex capture group> ]
```

```
      <right paren>

<regex position start or after> ::=
    START
  | AFTER

<regex occurrence> ::=
  <numeric value expression>

<regex capture group> ::=
  <numeric value expression>

<character position expression> ::=
  POSITION <left paren> <character value expression 1> IN <character value expression 2>
      [ USING <char length units> ] <right paren>

<character value expression 1> ::=
  <character value expression>

<character value expression 2> ::=
  <character value expression>

<binary position expression> ::=
  POSITION <left paren> <binary value expression> IN <binary value expression> <right paren>

<length expression> ::=
    <char length expression>
  | <octet length expression>

<char length expression> ::=
  { CHAR_LENGTH | CHARACTER_LENGTH } <left paren> <character value expression>
      [ USING <char length units> ] <right paren>

<octet length expression> ::=
  OCTET_LENGTH <left paren> <string value expression> <right paren>

<extract expression> ::=
  EXTRACT <left paren> <extract field> FROM <extract source> <right paren>

<extract field> ::=
    <primary datetime field>
  | <time zone field>

<time zone field> ::=
    TIMEZONE_HOUR
  | TIMEZONE_MINUTE

<extract source> ::=
    <datetime value expression>
  | <interval value expression>

<cardinality expression> ::=
  CARDINALITY <left paren> <collection value expression> <right paren>

<absolute value expression> ::=
  ABS <left paren> <numeric value expression> <right paren>

<modulus expression> ::=
  MOD <left paren> <numeric value expression dividend> <comma>
      <numeric value expression divisor><right paren>
```

```
<numeric value expression dividend> ::=
  <numeric value expression>

<numeric value expression divisor> ::=
  <numeric value expression>

<natural logarithm> ::=
  LN <left paren> <numeric value expression> <right paren>

<exponential function> ::=
  EXP <left paren> <numeric value expression> <right paren>

<power function> ::=
  POWER <left paren> <numeric value expression base> <comma>
      <numeric value expression exponent> <right paren>

<numeric value expression base> ::=
  <numeric value expression>

<numeric value expression exponent> ::=
  <numeric value expression>

<square root> ::=
  SQRT <left paren> <numeric value expression> <right paren>

<floor function> ::=
  FLOOR <left paren> <numeric value expression> <right paren>

<ceiling function> ::=
  { CEIL | CEILING } <left paren> <numeric value expression> <right paren>

<width bucket function> ::=
  WIDTH_BUCKET <left paren> <width bucket operand> <comma> <width bucket bound 1> <comma>
      <width bucket bound 2> <comma> <width bucket count> <right paren>

<width bucket operand> ::=
  <numeric value expression>

<width bucket bound 1> ::=
  <numeric value expression>

<width bucket bound 2> ::=
  <numeric value expression>

<width bucket count> ::=
  <numeric value expression>
```

Syntax Rules

1) If <position expression>, <regex occurrences function>, or <regex position expression> is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

2) If <character position expression> is specified, then <character value expression 1> and <character value expression 2> shall be comparable.

3) If <regex occurrences function> is specified, then:

   a) <XQuery pattern> and <regex subject string> shall be comparable.

b) The declared type of <start position> shall be exact numeric with scale 0 (zero).

c) If <start position> is not specified, then 1 (one) is implicit.

d) Case:

i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

ii) Otherwise, CHARACTERS is implicit.

e) If <XQuery option flag> is not specified, then the zero-length string is implicit.

4) If <regex position expression> is specified, then:

a) <XQuery pattern> and <regex subject string> shall be comparable.

b) If <regex position start or after> is not specified, then START is implicit.

c) The declared type of <start position>, <regex occurrence>, and <regex capture group> shall be exact numeric with scale 0 (zero).

d) If <start position> is not specified, then 1 (one) is implicit.

e) Case:

i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

ii) Otherwise, CHARACTERS is implicit.

f) If <regex occurrence> is not specified, then 1 (one) is implicit.

g) If <regex capture group> is not specified, then 0 (zero) is implicit.

h) If <XQuery option flag> is not specified, then the zero-length string is implicit.

5) Case:

a) If the character encoding form of <string value expression> is not UTF8, UTF16, or UTF32, then <char length units> shall not be specified.

b) Otherwise, if <char length units> is not specified, then CHARACTERS is implicit.

6) If <extract expression> is specified, then

Case:

a) If <extract field> is a <primary datetime field>, then it shall identify a <primary datetime field> of the <interval value expression> or <datetime value expression> immediately contained in <extract source>.

b) If <extract field> is a <time zone field>, then the declared type of the <extract source> shall be TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE.

7) If <extract expression> is specified, then

Case:

a) If <primary datetime field> does not specify SECOND, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

b) Otherwise, the declared type of the result is an implementation-defined exact numeric type with scale not less than the specified or implied <time fractional seconds precision> or <interval fractional seconds precision>, as appropriate, of the SECOND <primary datetime field> of the <extract source>.

8) If a <length expression> is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

9) If <cardinality expression> is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

10) If <absolute value expression> is specified, then the declared type of the result is the declared type of the immediately contained <numeric value expression>.

11) If <modulus expression> is specified, then the declared type of each <numeric value expression> shall be exact numeric with scale 0 (zero). The declared type of the result is the declared type of the immediately contained <numeric value expression divisor>.

12) The declared type of the result of <natural logarithm> is an implementation-defined approximate numeric type.

13) The declared type of the result of <exponential function> is an implementation-defined approximate numeric type.

14) The declared type of the result of <power function> is an implementation-defined approximate numeric type.

15) If <square root> is specified, then let *NVE* be the simply contained <numeric value expression>. The <square root> is equivalent to

```
POWER (NVE, 0.5)
```

16) If <floor function> or <ceiling function> is specified, then

Case:

a) If the declared type of the simply contained <numeric value expression> *NVE* is exact numeric, then the declared type of the result is exact numeric with implementation-defined precision, with the radix of *NVE*, and with scale 0 (zero).

b) Otherwise, the declared type of the result is approximate numeric with implementation-defined precision.

17) If <width bucket function> is specified, then the declared type of <width bucket count> shall be exact numeric with scale 0 (zero). The declared type of the result of <width bucket function> is the declared type of <width bucket count>.

Access Rules

None.

General Rules

1) If the value of one or more <string value expression>s, <datetime value expression>s, <interval value expression>s, and <collection value expression>s that are simply contained in a <numeric value function> is the null value, then the result of the <numeric value function> is the null value.

2) If <character position expression> is specified, then let *CVE1* be the value of <character value expression 1> and let *CVE2* be the value of <character value expression 2>.

   Case:

   a) If CHAR_LENGTH(*CVE1*) is 0 (zero), then the result is 1 (one).

   b) If <char length units> is specified, then let *CLU* be <char length units>; otherwise, let *CLU* be CHARACTERS. If there is at least one value *P* such that

      ```
      CVE1 = SUBSTRING ( CVE2 FROM P FOR CHAR_LENGTH (CVE1 USING CLU ) USING CLU )
      ```

      then the result is the least such *P*.

      NOTE 125 — The collation used is determined in the normal way.

   c) Otherwise, the result is 0 (zero).

3) If <binary position expression> is specified, then

   Case:

   a) If the first <binary value expression> has a length of 0 (zero), then the result is 1 (one).

   b) If the value of the first <binary value expression> is equal to an identical-length substring of contiguous octets from the value of the second <binary value expression>, then the result is 1 (one) greater than the number of octets within the value of the second <binary value expression> preceding the start of the first such substring.

   c) Otherwise, the result is 0 (zero).

4) If <extract expression> is specified, then

   Case:

   a) If <extract field> is a <primary datetime field>, then the result is the value of the datetime field identified by that <primary datetime field> and has the same sign as the <extract source>.

      NOTE 126 — If the value of the identified <primary datetime field> is zero or if <extract source> is not an <interval value expression>, then the sign is irrelevant.

   b) Otherwise, let *TZ* be the interval value of the implicit or explicit time zone displacement associated with the <datetime value expression>.

      Case:

      i) If <extract field> is TIMEZONE_HOUR, then the result is calculated as `EXTRACT (HOUR FROM TZ)`.

      ii) Otherwise, the result is calculated as `EXTRACT (MINUTE FROM TZ)`

5) If a <char length expression> is specified, then

   Case:

   a) If the character encoding form of <character value expression> is not UTF8, UTF16, or UTF32, then let *S* be the <string value expression>.

      Case:

i) If the most specific type of *S* is character string, then the result is the number of characters in the value of *S*.

NOTE 127 — The number of characters in a character string is determined according to the semantics of the character set of that character string.

ii) Otherwise, the result is OCTET_LENGTH(*S*).

b) Otherwise, the result is the number of explicit or implicit <char length units> in <char length expression>, counted in accordance with the definition of those units in the relevant normatively referenced document.

6) If an <octet length expression> is specified, then let *S* be the <string value expression>. Let *BL* be the number of bits (binary digits) in the value of *S*. The result of the <octet length expression> is the smallest integer not less than the quotient of the division (*BL*/8).

7) The result of <cardinality expression> is the number of elements of the result of the <collection value expression>.

8) If <absolute value expression> is specified, then let *N* be the value of the immediately contained <numeric value expression>.

Case:

a) If *N* is the null value, then the result is the null value.

b) If $N \geq 0$, then the result is *N*.

c) Otherwise, the result is $-1 * N$. If $-1 * N$ is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range*.

9) If <modulus expression> is specified, then let *N* be the value of the immediately contained <numeric value expression dividend> and let *M* be the value of the immediately contained <numeric value expression divisor>.

Case:

a) If either *N* or *M* is the null value, then the result is the null value.

b) If *M* is zero, then an exception condition is raised: *data exception — division by zero*.

c) Otherwise, the result is the unique exact numeric value *R* with scale 0 (zero) such that all of the following are true:

i) *R* has the same sign as *N*.

ii) The absolute value of *R* is less than the absolute value of *M*.

iii) $N = M * K + R$ for some exact numeric value *K* with scale 0 (zero).

10) If <natural logarithm> is specified, then let *V* be the value of the simply contained <numeric value expression>.

Case:

a) If *V* is the null value, then the result is the null value.

b) If *V* is 0 (zero) or negative, then an exception condition is raised: *data exception — invalid argument for natural logarithm*.

c) Otherwise, the result is the natural logarithm of *V*.

11) If <exponential function> is specified, then let *V* be the value of the simply contained <numeric value expression>.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise, the result is *e* (the base of natural logarithms) raised to the power *V*. If the result is not representable in the the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range*.

12) If <power function> is specified, then let *NVEB* be the <numeric value expression base>, then let *VB* be the value of *NVEB*, let *NVEE* be the <numeric value expression exponent>, and let *VE* be the value of *NVEE*.

Case:

a) If either *VB* or *VE* is the null value, then the result is the null value.

b) If *VB* is 0 (zero) and *VE* is negative, then an exception condition is raised: *data exception — invalid argument for power function*.

c) If *VB* is 0 (zero) and *VE* is 0 (zero), then the result is 1 (one).

d) If *VB* is 0 (zero) and *VE* is positive, then the result is 0 (zero).

e) If *VB* is negative and *VE* is not equal to an exact numeric value with scale 0 (zero), then an exception condition is raised: *data exception — invalid argument for power function*.

f) If *VB* is negative and *VE* is equal to an exact numeric value with scale 0 (zero) that is an even number, then the result is the result of

```
EXP(NVEE*LN(-NVEB))
```

g) If *VB* is negative and *VE* is equal to an exact numeric value with scale 0 (zero) that is an odd number, then the result is the result of

```
-EXP(NVEE*LN(-NVEB))
```

h) Otherwise, the result is the result of

```
EXP(NVEE*LN(NVEB))
```

13) If <floor function> is specified, then let *V* be the value of the simply contained <numeric value expression> *NVE*.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise,

Case:

i) If the most specific type of *NVE* is exact numeric, then the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to *V*. If this result is not representable by the

result data type, then an exception condition is raised: *data exception — numeric value out of range*.

ii) Otherwise, the result is the greatest whole number that is less than or equal to *V*. If this result is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range*.

14) If <ceiling function> is specified, then let *V* be the value of the simply contained <numeric value expression> *NVE*.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise,

Case:

i) If the most specific type of *NVE* is exact numeric, then the result is the least exact numeric value with scale 0 (zero) that is greater than or equal to *V*. If this result is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range*.

ii) Otherwise, the result is the least whole number that is greater than or equal to *V*. If this result is not representable by the result data type, then an exception condition is raised: *data exception — numeric value out of range*.

15) If <width bucket function> is specified, then let *WBO* be the value of <width bucket operand>, let *WBB1* be the value of <width bucket bound 1>, let *WBB2* be the value of <width bucket bound 2>, and let *WBC* be the value of <width bucket count>.

Case:

a) If any of *WBO*, *WBB1*, *WBB2*, or *WBC* is the null value, then the result is the null value.

b) If *WBC* is less than or equal to 0 (zero), then an exception condition is raised: *data exception — invalid argument for width bucket function*.

c) If *WBB1* equals *WBB2*, then an exception condition is raised: *data exception — invalid argument for width bucket function*.

d) If *WBB1* is less than *WBB2*, then

Case:

i) If *WBO* is less than *WBB1*, then the result is 0 (zero).

ii) If *WBO* is greater than or equal to *WBB2*, then the result is *WBC*+1. If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range*.

iii) Otherwise, the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to `((WBC * (WBO - WBB1) / (WBB2 - WBB1)) + 1)`

e) If *WBB1* is greater than *WBB2*, then

Case:

i) If *WBO* is greater than *WBB1*, then the result is 0 (zero).

ii) If *WBO* is less than or equal to *WBB2*, then the result is *WBC*+1. If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range*.

iii) Otherwise, the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to `((WBC * (WBB1 - WBO) / (WBB1 - WBB2)) + 1)`

16) If <regex occurrences function> is specified, then

a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, and let *FL* be the value of <XQuery option flag>.

b) Case:

i) If *STR*, *PAT*, or *FL* is the null value, then the result of the <regex occurrences function> is the null value.

ii) If *SP* is less than 1 (one), or greater than the value of

```
CHARACTER_LENGTH ( RSS USING CLU )
```

then the result of <regex occurrences function> is –1.

iii) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex occurrences function> is implementation-dependent.

iv) Otherwise, let *LIST* be the list of match vectors that is the result of applying the General Rules Subclause 9.16, "XQuery regular expression matching", with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, and *FL* as *FLAG*. The result of <regex occurrences function> is the number of match vectors in LIST.

17) If <regex position expression> is specified, then:

a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, let *OCC* be the value of <regex occurrence>, let *CAP* be the <regex capture group> and let *FL* be the value of <XQuery option flag>.

b) Case:

i) If *STR*, *PAT*, *OCC*, *CAP*, or *FL* is the null value, then the result of the <regex position expression> is the null value.

ii) If *OCC* is less than 1 (one), then the result of <regex position expression> is 0 (zero).

iii) If *CAP* is less than 0 (zero), or greater than the number of XQuery regular expression parenthesized sub-expressions of *PAT*, then the result of <regex position expression> is 0 (zero).

iv) If *SP* is less than 1 (one), or greater than the value of

```
CHARACTER_LENGTH ( RSS USING CLU )
```

then the result of <regex position expression> is 0 (zero).

v) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex position expression> is implementation-dependent.

vi) Otherwise, let *LIST* be the list of match vectors that is the result of applying the General Rules of Subclause 9.16, "XQuery regular expression matching", with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, and *FL* as *FLAG*.

Case:

1) If there are at least *OCC* match vectors in *LIST*, then let *MV* be the *OCC*-th match vector in *LIST*. Let *PL* be *MV*[*CAP*], where *MV*[0] is the first position/length in *MV*. Let *P* be the position of *PL*, and let *L* be the length of *PL*. The result of the <regex position expression> is

Case:

A) If *P* is 0 (zero), then 0 (zero).

B) If <regex position start or after> is START, then *P*.

C) Otherwise, *P* + *L*.

2) Otherwise, the result of the <regex position expression> is 0 (zero).

Conformance Rules

1) Without Feature S091, "Basic array support", or Feature S271, "Basic multiset support", conforming SQL language shall not contain a <cardinality expression>.

2) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <extract expression>.

3) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <extract expression> that specifies a <time zone field>.

4) Feature F411, "Time zone specification", conforming SQL language shall not contain an <extract expression> that specifies a <time zone field>.

5) Without Feature F421, "National character", conforming SQL language shall not contain a <length expression> that simply contains a <string value expression> that has a declared type of NATIONAL CHARACTER LARGE OBJECT.

6) Without Feature T441, "ABS and MOD functions", conforming language shall not contain an <absolute value expression>.

7) Without Feature T441, "ABS and MOD functions", conforming language shall not contain a <modulus expression>.

8) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <natural logarithm>.

9) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain an <exponential function>.

10) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <power function>.

11) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <square root>.

12) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <floor function>.

13) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <ceiling function>.

14) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <width bucket function>.

15) Without Feature F842, “OCCURENCES_REGEX function”, conforming SQL language shall not contain <regex occurrences function>.

16) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language, <regex occurrences function> shall not contain <char length units>.

17) Without Feature F843, “POSITION_REGEX function”, conforming SQL language shall not contain <regex position expression>.

18) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language , <regex position expression> shall not contain <char length units>.

19) Without Feature F847, “Nonconstant regular expression”, in conforming SQL language, <XQuery pattern> and <XQuery option flag> shall be <value specification>s.

6.28 <string value expression>

Function

Specify a character string value or a binary string value.

Format

```
<string value expression> ::=
    <character value expression>
  | <binary value expression>

<character value expression> ::=
    <concatenation>
  | <character factor>

<concatenation> ::=
  <character value expression> <concatenation operator> <character factor>

<character factor> ::=
  <character primary> [ <collate clause> ]

<character primary> ::=
    <value expression primary>
  | <string value function>

<binary value expression> ::=
    <binary concatenation>
  | <binary factor>

<binary factor> ::=
  <binary primary>

<binary primary> ::=
    <value expression primary>
  | <string value function>

<binary concatenation> ::=
  <binary value expression> <concatenation operator> <binary factor>
```

Syntax Rules

1) The declared type of a <character primary> shall be character string.

2) The declared type of a <character value expression> is the declared type of the simply contained <concatenation> or <character factor>.

3) If <concatenation> is specified, then:

   a) Let *D1* be the declared type of the <character value expression> and let *D2* be the declared type of the <character factor> simply contained in the <concatenation>. Let *D* be the data type determined by applying the Syntax Rules of Subclause 9.3, "Result of data type combinations", to *D1* and *D2*. Let *CS* be the character set of *D*, let *CO* be the collation of *D*, and let *CD* be the collation derivation of *D*.

b) Let *M* be the length in characters of *D1* plus the length in characters of *D2*. Let *VL* be the implementation-defined maximum length of variable-length character strings, let *LOL* be the implementation-defined maximum length of large object character strings, and let *FL* be the implementation-defined maximum length of fixed-length character strings.

Case:

i) If the declared type of the <character value expression> or <character factor> is a character large object type, then the declared type of the <concatenation> is a character large object type with character set *CS*, collation *CO*, collation derivation *CD* and maximum length equal to the lesser of *M* and *LOL*.

ii) If the declared type of the <character value expression> or <character factor> is variable-length character string, then the declared type of the <concatenation> is variable-length character string with character set *CS*, collation *CO*, collation derivation *CD* and maximum length equal to the lesser of *M* and *VL*.

iii) If the declared type of the <character value expression> and <character factor> is fixed-length character string, then *M* shall not be greater than *FL* and the declared type of the <concatenation> is fixed-length character string with character set *CS*, collation *CO*, collation derivation *CD* and length *M*.

4) If <character factor> is specified, then

Case:

a) If <collate clause> is specified, then the declared type of the <character factor> is the declared type of the <character primary>, except that the declared type collation is the collation identified by <collate clause>, and its collation derivation is *explicit*.

b) Otherwise, the declared type of the <character factor> is the declared type of the <character primary>.

5) The declared type of <binary primary> shall be binary string.

6) Case:

a) If <binary concatenation> is specified, then let *B1* be the declared type of the <binary value expression> and let *B2* be the declared type of the <binary factor>. Let *M* be the length in octets of *B1* plus the length in octets of *B2*. Let *FL* be the implementation-defined maximum length of fixed-length binary strings, let *VL* be the implementation-defined maximum length of variable-length binary strings, let *LOL* be the implementation-defined maximum length of binary large object strings.

The declared type of <binary concatenation> is

Case:

i) If the declared type of the <binary value expression> or <binary factor> is a binary large object type, then binary large object string with maximum length equal to the lesser of *M* and *LOL*.

ii) If the declared type of the <binary value expression> or <binary factor> is variable-length binary string, then variable-length binary string with maximum length equal to the lesser of *M* and *VL*.

iii) If the declared type of the <binary value expression> and <binary factor> is fixed-length binary string, then fixed-length binary string with length *M*, and *M* shall not be greater than *FL*.

b) Otherwise, the declared type of the <binary value expression> is the declared type of the <binary factor>.

Access Rules

None.

General Rules

1) If the value of any <character primary> or <binary primary> simply contained in a <string value expression> is the null value, then the result of the <string value expression> is the null value.

2) If <concatenation> is specified, then:

   a) If the character repertoire of <character factor> is UCS, then, in the remainder of this General Rule, the term "length" shall be taken to mean "length in characters".

   b) Let *S1* and *S2* be the result of the <character value expression> and <character factor>, respectively.

      Case:

      i) If either *S1* or *S2* is the null value, then the result of the <concatenation> is the null value.

      ii) Otherwise:

         1) Let *S* be the string consisting of *S1* followed by *S2* and let *M* be the length of *S*.

         2) If the character repertoire of <character factor> is UCS, then *S* is replaced by

            Case:

            A) If the <search condition> `S1 IS NORMALIZED AND S2 IS NORMALIZED` evaluates to *True*, then

               `NORMALIZE (S)`

            B) Otherwise, an implementation-defined string.

         3) Case:

            A) If the most specific type of either *S1* or *S2* is a character large object type, then let *LOL* be the implementation-defined maximum length of large object character strings.

               Case:

               I) If *M* is less than or equal to *LOL*, then the result of the <concatenation> is *S* with length *M*.

               II) If *M* is greater than *LOL* and the right-most *M–LOL* characters of *S* are all the <space> character, then the result of the <concatenation> is the first *LOL* characters of *S* with length *LOL*.

               III) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

            B) If the most specific type of either *S1* or *S2* is variable-length character string, then let *VL* be the implementation-defined maximum length of variable-length character strings.

               Case:

I) If *M* is less than or equal to *VL*, then the result of the <concatenation> is *S* with length *M*.

II) If *M* is greater than *VL* and the right-most *M–VL* characters of *S* are all the <space> character, then the result of the <concatenation> is the first *VL* characters of *S* with length *VL*.

III) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

C) If the most specific types of both *S1* and *S2* are fixed-length character string, then the result of the <concatenation> is *S*.

3) If <binary concatenation> is specified, then let *S1* and *S2* be the result of the <binary value expression> and <binary factor>, respectively.

Case:

a) If either *S1* or *S2* is the null value, then the result of the <binary concatenation> is the null value.

b) Otherwise, let *S* be the string consisting of *S1* followed by *S2* and let *M* be the length in octets of *S*.

Case:

i) If the most specific type of either *S1* or *S2* is a binary large object type, then let *LOL* be the implementation-defined maximum length of binary large object strings.

Case:

1) If *M* is less or equal to *LOL*, then the result of the <binary concatenation> is *S* with length *M*.

2) If *M* is greater than *LOL* and the right-most *M–LOL* octets of *S* are all X'00', then the result of the <binary concatenation> is the first *LOL* octets of *S* with length *VL*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

ii) If If the most specific type of either *S1* or *S2* is variable-length binary string, then let *VL* be the implementation-defined maximum length of variable-length binary strings.

Case:

1) If *M* is less than or equal to *VL*, then the result of the <binary concatenation> is *S* with length *M*.

2) If *M* is greater than *VL* and the right-most *M–VL* octets of *S* are all X'00', then the result of the <binary concatenation> is the first *VL* characters of *S* with length *VL*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

iii) If the most specific types of both *S1* and *S2* are fixed-length binary string, then the result of the <binary concatenation> is *S*.

4) If the result of the <character value expression> is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string*.

Conformance Rules

None.

6.29 <string value function>

This Subclause is modified by Subclause 6.4, "<string value function>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.8, "<string value function>", in ISO/IEC 9075-14.

Function

Specify a function yielding a value of type character string or binary string.

Format

```
<string value function> ::=
    <character value function>
  | <binary value function>

<character value function> ::=
    <character substring function>
  | <regular expression substring function>
  | <regex substring function>
  | <fold>
  | <transcoding>
  | <character transliteration>
  | <regex transliteration>
  | <trim function>
  | <character overlay function>
  | <normalize function>
  | <specific type method>

<character substring function> ::=
  SUBSTRING <left paren> <character value expression> FROM <start position>
      [ FOR <string length> ] [ USING <char length units> ] <right paren>

<regular expression substring function> ::=
  SUBSTRING <left paren> <character value expression> SIMILAR <character value expression>
      ESCAPE <escape character> <right paren>

<regex substring function> ::=
  SUBSTRING_REGEX <left paren>
      <XQuery pattern> [ FLAG <XQuery option flag> ]
      IN <regex subject string>
      [ FROM <start position> ]
      [ USING <char length units> ]
      [ OCCURRENCE <regex occurrence> ]
      [ GROUP <regex capture group> ]
      <left paren>

<fold> ::=
  { UPPER | LOWER } <left paren> <character value expression> <right paren>

<transcoding> ::=
  CONVERT <left paren> <character value expression>
      USING <transcoding name> <right paren>

<character transliteration> ::=
  TRANSLATE <left paren> <character value expression>
```

```
      USING <transliteration name> <right paren>

<regex transliteration> ::=
  TRANSLATE_REGEX <left paren>
      <XQuery pattern> [ FLAG <XQuery option flag> ]
      IN <regex subject string>
      [ WITH <XQuery replacement string> ]
      [ FROM <start position> ]
      [ USING <char length units> ]
      [ OCCURRENCE <regex transliteration occurrence> ]
      <left paren>

<XQuery replacement string> ::=
  <character value expression>

<regex transliteration occurrence> ::=
    <regex occurrence>
  | ALL

<trim function> ::=
  TRIM <left paren> <trim operands> <right paren>

<trim operands> ::=
  [ [ <trim specification> ] [ <trim character> ] FROM ] <trim source>

<trim source> ::=
  <character value expression>

<trim specification> ::=
    LEADING
  | TRAILING
  | BOTH

<trim character> ::=
  <character value expression>

<character overlay function> ::=
  OVERLAY <left paren> <character value expression> PLACING <character value expression>
      FROM <start position> [ FOR <string length> ]
      [ USING <char length units> ] <right paren>

<normalize function> ::=
  NORMALIZE <left paren> <character value expression>
      [ <comma> <normal form> [ <normalize function result length> ] ] <right paren>

<normal form> ::=
    NFC
  | NFD
  | NFKC
  | NFKD

<normalize function result length> ::=
    <length>
  | <large object length>

<specific type method> ::=
  <user-defined type value expression> <period> SPECIFICTYPE
      [ <left paren> <right paren> ]

<binary value function> ::=
```

```
    <binary substring function>
  | <binary trim function>
  | <binary overlay function>

<binary substring function> ::=
  SUBSTRING <left paren> <binary value expression> FROM <start position>
      [ FOR <string length> ] <right paren>

<binary trim function> ::=
  TRIM <left paren> <binary trim operands> <right paren>

<binary trim operands> ::=
  [ [ <trim specification> ] [ <trim octet> ] FROM ] <binary trim source>

<binary trim source> ::=
  <binary value expression>

<trim octet> ::=
  <binary value expression>

<binary overlay function> ::=
  OVERLAY <left paren> <binary value expression> PLACING <binary value expression>
      FROM <start position> [ FOR <string length> ] <right paren>

<start position> ::=
  <numeric value expression>

<string length> ::=
  <numeric value expression>
```

Syntax Rules

1) The declared type of <string value function> is the declared type of the immediately contained <character value function> or <binary value function>.

2) The declared type of <character value function> is the declared type of the immediately contained <character substring function>, <regular expression substring function>, <regex substring function>, <fold>, <transcoding>, <character transliteration>, <regex transliteration>, <trim function>, <character overlay function>, <normalize function>, or <specific type method>.

3) The declared type of a <start position>, <string length>, <regex occurrence>, or <regex capture group> shall be exact numeric with scale 0 (zero).

4) If <character substring function> *CSF* is specified, then let *DTCVE* be the declared type of the <character value expression> immediately contained in *CSF*. The maximum length, character set, and collation of the declared type *DTCSF* of *CSF* are determined as follows:

   a) Case:

      i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then *DTCSF* is a variable-length character string type with maximum length equal to the fixed length or maximum length of *DTCVE*.

      ii) Otherwise, the *DTCSF* is a large object character string type with maximum length equal to the maximum length of *DTCVE*.

   b) The character set and collation of the <character substring function> are those of *DTCVE*.

5) If the character repertoire of <character value expression> is not UCS, then <char length units> shall not be specified.

6) If USING <char length units> is not specified, then USING CHARACTERS is implicit.

7) If <regular expression substring function> is specified, then:

   a) The declared types of the <escape character> and the <character value expression>s of the <regular expression substring function> shall be character string with the same character repertoire.

   b) Case:

      i) If the declared type of the first <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <regular expression substring function> is variable-length character string with maximum length equal to the maximum variable length of the first <character value expression>.

      ii) Otherwise, the declared type of the <regular expression substring function> is a character large object type with maximum length equal to the maximum variable length of the first <character value expression>.

   c) The declared type of the <regular expression substring function> is that of the first <character value expression>.

8) If <regex substring function> *RSF* is specified, then:

   a) If <start position> is not specified, then 1 (one) is implicit.

   b) Case:

      i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

      ii) Otherwise, CHARACTERS is implicit.

   c) If <regex occurrence> is not specified, then 1 (one) is implicit.

   d) If <regex capture group> is not specified, then 0 (zero) is implicit.

   e) If <XQuery option flag> is not specified, then the zero-length string is implicit.

   f) Let *RSS* be the <regex subject string> immediately contained in *RSF*. Let *DTCVE* be the declared type of *RSS*. The declared type *DTRSF* of *RSF* is determined as follows:

      i) Case:

         1) If *DTCVE* is fixed-length character string or variable-length character string, then *DTRSF* is variable length character string with maximum length equal to the fixed length or maximum length of *DTCVE*.

         2) Otherwise, *DTRSF* is a large object character string type with maximum length equal to the maximum length of *DTCVE*.

      ii) The character set and collation of *DTRSF* are those of *DTCVE*.

9) If <fold> is specified, then the declared type of the result of <fold> is that of the <character value expression>.

10) If <transcoding> is specified, then:

a) <transcoding> shall be simply contained in a <value expression> that is immediately contained in a <derived column> that is immediately contained in a <select sublist> or shall immediately contain either a <simple value specification> that is a <host parameter name> or a <value specification> that is a <host parameter specification>.

b) A <transcoding name> shall identify a transcoding.

c) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the result is variable-length character string with implementation-defined maximum length.

ii) Otherwise, the declared type of the result is a character large object type with implementation-defined maximum length.

d) The character set of the result is an implementation-defined character set *CS* whose character repertoire is the same as the character repertoire of the <character value expression> and whose character encoding form is that determined by the transcoding identified by the <transcoding name>. The declared type collation of the result is the character set collation of *CS*.

11) If <character transliteration> is specified, then:

a) A <transliteration name> shall identify a character transliteration.

b) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <character transliteration> is variable-length character string with implementation-defined maximum length.

ii) Otherwise, the declared type of the <character transliteration> is a character large object type with implementation-defined maximum length.

c) The declared type of the <character transliteration> has the character set *CS* that is the target character set of the transliteration. The declared type collation of the result is the character set collation of *CS*.

12) If <regex transliteration> *RT* is specified, then:

a) If <start position> is not specified, then 1 (one) is implicit.

b) Case:

i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

ii) Otherwise, CHARACTERS is implicit.

c) If <regex transliteration occurrence> is not specified, then ALL is implicit.

d) If <XQuery replacement string> is not specified, then the zero-length string is implicit.

e) If <XQuery option flag> is not specified, then the zero-length string is implicit.

f) Let *RSS* be the <regex subject string> immediately contained in *RT*. Let *DTCVE* be the declared type of *RSS*. The declared type *DTRT* of *RT* is determined as follows:

i) Case:

1) If <XQuery replacement string> is not specified, or is a zero-length string, then:

   A) If *DTCVE* is fixed-length character string or variable-length character string, then *DTRT* is variable length character string with maximum length equal to the fixed length or maximum length of *DTCVE*.

   B) Otherwise, *DTRT* is a large object character string type with maximum length equal to the maximum length of *DTCVE*.

2) Otherwise:

   A) If *DTCVE* is fixed-length character string or variable-length character string, then *DTRT* is variable length character string with implementation-defined maximum length.

   B) Otherwise, *DTRT* is a large object character string type with implementation-defined maximum length.

ii) The character set and collation of *DTRT* are those of *DTCVE*.

13) If <trim function> is specified, then

a) Case:

i) If FROM is specified, then:

1) Either <trim specification> or <trim character> or both shall be specified.

2) If <trim specification> is not specified, then BOTH is implicit.

3) If <trim character> is not specified, then `' '` is implicit.

ii) Otherwise, let *SRC* be <trim source>. `TRIM ( SRC )` is equivalent to `TRIM ( BOTH ' ' FROM SRC )`.

b) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <trim function> is variable-length character string with maximum length equal to the fixed length or maximum variable length of the <trim source>.

ii) Otherwise, the declared type of the <trim function> is a character large object type with maximum length equal to the maximum variable length of the <trim source>.

c) If a <trim character> is specified, then <trim character> and <trim source> shall be comparable.

d) The declared type of the <trim function> is that of the <trim source>.

14) If <character overlay function> is specified, then:

a) Let *CV* be the first <character value expression>, let *SP* be the <start position>, and let *RS* be the second <character value expression>.

b) If <string length> is specified, then let *SL* be <string length>; otherwise, let *SL* be CHAR_LENGTH(*RS*).

c) The <character overlay function> is equivalent to:

```
SUBSTRING ( CV FROM 1 FOR SP - 1 )
```

```
|| RS
|| SUBSTRING ( CV FROM SP + SL  )
```

15) If <normalize function> is specified, then:

a) Case:

i) If <normal form> is specified, then let *NF* be <normal form>.

ii) Otherwise, let *NF* be NFC.

b) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then:

1) <large object length> shall not be specified.

2) If <normalize function result length> is specified, then let *L* be <normalize function result length>; otherwise, let *L* be an implementation-defined value of exact numeric type with scale 0 (zero) that is less than or equal to the implementation-defined maximum length of variable-length character strings.

3) The declared type of the <normalize function> is variable-length character string with maximum length equal to *L*.

ii) Otherwise:

1) If <normalize function result length> is specified, then let *L* be the <normalize function result length>; otherwise, let *L* be an implementation-defined value of exact numeric type with scale 0 (zero) that is less than or equal to the implementation-defined maximum length of large object character strings.

2) The declared type of the <normalize function> is character large object type with maximum length equal to *L*.

16) If <specific type method> is specified, then the declared type of the <specific type method> is variable-length character string with maximum length implementation-defined. The character set of the character string is SQL_IDENTIFIER.

17) The declared type of <binary value function> is the declared type of the immediately contained <binary substring function>, <binary trim function>, or <binary overlay function>.

18) If <binary substring function> *BSF* is specified, then let *DTBVE* be the declared type of the <binary value expression> immediately contained in *BSF*. The declared type *DTBSF* of *BSF* and its maximum length are

Case:

a) If *DTBVE* is fixed-length binary string or variable-length binary string, then variable-length binary string type with maximum length equal to the fixed or maximum variable length of *DTBVE*.

b) Otherwise, binary large object string type with maximum length equal to the maximum length of the <binary value expression>.

19) If <binary trim function> is specified, then:

a) Case:

i) If FROM is specified, then:

1) Either <trim specification> or <trim octet> or both shall be specified.

2) If <trim specification> is not specified, then BOTH is implicit.

3) If <trim octet> is not specified, then X'00' is implicit.

ii) Otherwise, let *SRC* be <trim source>. `TRIM ( SRC )` is equivalent to `TRIM ( BOTH X'00' FROM SRC )`.

b) Case:

i) If the declared type of <binary value expression> is fixed-length binary string or variable-length binary string, then the declared type of the <binary trim function> is variable-length binary string type with maximum length equal to the fixed or maximum variable length of the <binary trim source>.

ii) Otherwise, the declared type of the <binary trim function> is binary large object string type with maximum length equal to the maximum length of the <binary trim source>.

20) If <binary overlay function> is specified, then:

a) Let *BV* be the first <binary value expression>, let *SP* be the <start position>, and let *RS* be the second <binary value expression>.

b) If <string length> is specified, then let *SL* be <string length>; otherwise, let *SL* be OCTET_LENGTH(*RS*).

c) The <binary overlay function> is equivalent to:

```
   SUBSTRING ( BV FROM 1 FOR SP - 1 )
|| RS
|| SUBSTRING ( BV FROM SP + SL  )
```

Access Rules

1) Case:

a) If <string value function> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE for every transliteration identified by a <transliteration name> contained in the <string value expression>.

b) Otherwise, the current privileges shall include USAGE for every transliteration identified by a <transliteration name> contained in the <string value expression>.

General Rules

1) The result of <string value function> is the result of the immediately contained <character value function> or <binary value function>.

2) The result of <character value function> is the result of the immediately contained <character substring function>, <regular expression substring function>, <regex substring function>, <fold>, <transcoding>,

<character transliteration>, <regex transliteration>, <trim function>, <character overlay function>, <normalize function>, or <specific type method>.

3) If <character substring function> is specified, then:

   a) If the character encoding form of <character value expression> is UTF8, UTF16, or UTF32, then, in the remainder of this General Rule, the term "character" shall be taken to mean "unit specified by <char length units>".

   b) Let *C* be the value of the <character value expression>, let *LC* be the length in characters of *C*, and let *S* be the value of the <start position>.

   c) If <string length> is specified, then let *L* be the value of <string length> and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.

   d) If either *C*, *S*, or *L* is the null value, then the result of the <character substring function> is the null value.

   e) If *E* is less than *S*, then an exception condition is raised: *data exception — substring error*.

   f) Case:

      i) If *S* is greater than *LC* or if *E* is less than 1 (one), then the result of the <character substring function> is a zero-length string.

      ii) Otherwise,

         1) Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*–*S1*.

         2) The result of the <character substring function> is a character string containing the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

4) If <regex substring function> is specified, then:

   a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, let *OCC* be the value of <regex occurrence>, let *CAP* be the <regex capture group>, and let *FL* be the value of <XQuery option flag>.

   b) Case:

      i) If *STR*, *PAT*, *OCC*, *CAP*, or *FL* is the null value, then the result of the <regex substring function> is the null value.

      ii) If *OCC* is less than 1 (one), then the result of <regex substring function> is the null value.

      iii) If *CAP* is less than 0 (zero), or greater than the number of XQuery regular expression parenthesized sub-expressions of *PAT*, then the result of <regex substring function> is the null value.

      iv) If *SP* is less than 1 (one) or greater than the value of

         ```
         CHARACTER_LENGTH ( RSS USING CLU )
         ```

         then the result of <regex substring function> is the null value.

      v) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex substring function> is implementation-dependent.

vi) Let *LIST* be the list of match vectors that is the result of applying the General Rules of Subclause 9.16, "XQuery regular expression matching", with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, *FL* as *FLAG*.

Case:

1) If there are at least *OCC* match vectors in *LIST*, then let *MV* be the *OCC-th* position/length. Let *PL* be *MV*[*CAP*], where *MV*[0] is the first position length in *MV*. Let *P* be the position of *PL* and let *L* be the length of *PL*. The result of the <regex substring function> is:

A) If *P* is 0 (zero), then the null value.

B) Otherwise, the substring of *STR* whose position is *P* and whose length is *L*.

2) Otherwise, the result of the <regex substring function> is the null value.

5) If <normalize function> is specified, then:

a) Let *S* be the value of <character value expression>.

b) If *S* is the null value, then the result of the <normalize function> is the null value.

c) Let *NR* be *S* in the normalized form specified by *NF* in accordance with [UNICODE15].

d) Case:

i) If the length in characters of *NR* is less than or equal to *L*, then the result of the <normalize function> is *NR*.

ii) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

6) If <regular expression substring function> is specified, then:

a) Let *C* be the result of the first <character value expression>, let *R* be the result of the second <character value expression>, and let *E* be the result of the <escape character>.

b) If one or more of *C*, *R*, or *E* is the null value, then the result of the <regular expression substring function> is the null value.

c) If the length in characters of *E* is not equal to 1 (one), then an exception condition is raised: *data exception — invalid escape character*.

d) If *R* does not contain exactly two occurrences of the two-character sequence consisting of *E*, each immediately followed by <double quote>, then an exception condition is raised: *data exception — invalid use of escape character*.

e) Let *R1*, *R2*, and *R3* be the substrings of *R*, such that

```
'R' = 'R1' || 'E' || '"' || 'R2' || 'E' || '"' || 'R3'
```

is *True*.

f) If any one of *R1*, *R2*, or *R3* is not a zero-length string and does not have the format of a <regular expression>, then an exception condition is raised: *data exception — invalid regular expression*.

g) If the predicate

```
'C' SIMILAR TO 'R1' || 'R2' || 'R3' ESCAPE 'E'
```

is not *True*, then the result of the <regular expression substring function> is the null value.

h) Otherwise, the result *S* of the <regular expression substring function> is computed as follows:

i) Let *S1* be the shortest initial substring of *C* such that there is a substring *S23* of *C* such that the value of the following <search condition> is *True*:

```
'C' = 'S1' || 'S23' AND
'S1' SIMILAR TO 'R1' ESCAPE 'E' AND
'S23' SIMILAR TO '(R2R3)' ESCAPE 'E'
```

ii) Let *S3* be the shortest final substring of *S23* such that there is a substring *S2* of *S23* such that the value of the following <search condition> is *True*:

```
'S23' = 'S2' || 'S3' AND
'S2' SIMILAR TO 'R2' ESCAPE 'E' AND
'S3' SIMILAR TO 'R3' ESCAPE 'E'
```

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

7) If <fold> is specified, then:

a) Let *S* be the value of the <character value expression>.

b) If *S* is the null value, then the result of the <fold> is the null value.

c) Let *FRML* be the length or maximum length in characters of the declared type of <fold>.

d) Case:

i) If UPPER is specified, then let *FR* be a copy of *S* in which every lower case character that has a corresponding upper case character or characters in the character set of *S* and every title case character that has a corresponding upper case character or characters in the character set of *S* is replaced by that upper case character or characters.

ii) If LOWER is specified, then let *FR* be a copy of *S* in which every upper case character that has a corresponding lower case character or characters in the character set of *S* and every title case character that has a corresponding lower case character or characters in the character set of *S* is replaced by that lower case character or characters.

e) If the character set of <character factor> is UTF8, UTF16, or UTF32, then *FR* is replaced by

Case:

i) If the <search condition> `S IS NORMALIZED` evaluated to *True*, then

```
NORMALIZE (FR)
```

ii) Otherwise, *FR*.

f) Let *FRL* be the length in characters of *FR*.

g) Case:

i) If *FRL* is less than or equal to *FRML*, then the result of the <fold> is *FR*. If the declared type of *FR* is fixed-length character string, then the result is padded on the right with (*FRML* – *FRL*) <space>s.

ii) If *FRL* is greater than *FRML*, then the result of the <fold> is the first *FRML* characters of *FR* with length *FRML*. If any of the right-most (*FRL* – *FRML*) characters of *FR* are not <space> characters, then a completion condition is raised: *warning — string data, right truncation*.

8) If a <character transliteration> is specified, then

Case:

a) If the value of <character value expression> is the null value, then the result of the <character transliteration> is the null value.

b) If <transliteration name> identifies a transliteration descriptor whose indication of how the transliteration is performed specifies an SQL-invoked routine *TR*, then the result of the <character transliteration> is the result of the invocation of *TR* with a single SQL argument that is the <character value expression> contained in the <character transliteration>.

c) Otherwise, the value of the <character transliteration> is the value returned by the transliteration identified by the <existing transliteration name> specified in the transliteration descriptor of the transliteration identified by <transliteration name>.

9) If <regex transliteration> is specified, then:

a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, let *REP* be the value of <XQuery replacement string>, and let *FL* be the value of <XQuery option flag>.

b) Case:

i) If *STR*, *PAT*, *REP*, or *FL* is the null value, then the result of the <regex transliteration> is the null value.

ii) If *SP* is less than 1 (one) or greater than the value of

```
CHARACTER_LENGTH ( RSS USING CLU )
```

then the result of <regex transliteration> is the null value.

iii) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex transliteration> is implementation-dependent.

iv) Otherwise, let *LIST* be the list of match vectors that is the result of applying the General Rules of Subclause 9.16, "XQuery regular expression matching", with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, *FL* as *FLAG*. Let *RTO* be the <regex transliteration occurrence>.

Case:

1) If *RTO* is <regex occurrence>, then let *OCC* be the value of *RTO*.

Case:

A) If *OCC* is the null value, then the result of the <regex transliteration> is the null value.

B) If *OCC* is less than 1 (one) or greater than the number of match vectors in *LIST*, then the result of the <regex transliteration> is *STR*.

C) Otherwise, let *MV* be the *OCC*-th match vector in *LIST*. The result of the <regex transliteration> is the result of applying the General Rules of Subclause 9.17, "XQuery

regular expression replacement", with *MV* as *MATCH*, *STR* as *STRING*, *PAT* as *PATTERN*, *REP* as *REPLACEMENT*, *FL* as *FLAG*.

2) Otherwise, <regex transliteration occurrence> is ALL. Let *N* be the number of match vectors in *LIST*. Let S_N be *STR*.

   A) For all *i* between *N* and 1(one) in descending order, let MV_i be the *i*-th match vector in *LIST*. Let S_{i-1} be the result of applying the General Rules of Subclause 9.17, "XQuery regular expression replacement", with MV_i as *MATCH*, S_i as *STRING*, *PAT* as *PATTERN*, *REP* as *REPLACEMENT*, *FL* as *FLAG*.

   NOTE 128 — The replacements are applied from the end of *STR* moving to the front to avoid altering the positions of matches within the string. When a replacement is performed, there are no unprocessed replacements later in the string, and consequently the positions of other matches are not disturbed if a substring is replaced by a string of a different length.

   B) The result of <regex transliteration> is S_0.

10) If a <transcoding> is specified, then

    Case:

    a) If the value of <character value expression> is the null value, then the result of the <transcoding> is the null value.

    b) Otherwise, the value of the <transcoding> is the value of the <character value expression> after the application of the transcoding specified by <transcoding name>.

11) If <trim function> is specified, then:

    a) Let *S* be the value of the <trim source>.

    b) If <trim character> is specified, then let *SC* be the value of <trim character>; otherwise, let *SC* be <space>.

    c) If either *S* or *SC* is the null value, then the result of the <trim function> is the null value.

    d) If the length in characters of *SC* is not 1 (one), then an exception condition is raised: *data exception — trim error*.

    e) Case:

       i) If BOTH is specified or if no <trim specification> is specified, then the result of the <trim function> is the value of *S* with any leading or trailing characters equal to *SC* removed.

       ii) If TRAILING is specified, then the result of the <trim function> is the value of *S* with any trailing characters equal to *SC* removed.

       iii) If LEADING is specified, then the result of the <trim function> is the value of *S* with any leading characters equal to *SC* removed.

12) If <specific type method> is specified, then:

    a) Let *V* be the value of the <user-defined type value expression>.

    b) Case:

       i) If *V* is the null value, then *RV* is the null value.

ii) Otherwise:

1) Let *UDT* be the most specific type of *V*.

2) Let *UDTN* be the <user-defined type name> of *UDT*.

3) Let *CN* be the <catalog name> contained in *UDTN*, let *SN* be the <unqualified schema name> contained in *UDTN*, and let *UN* be the <qualified identifier> contained in *UDTN*. Let *CND*, *SND*, and *UND* be *CN*, *SN*, and *UN*, respectively, with every occurrence of <double quote> replaced by <doublequote symbol>. Let *RV* be:

```
"CND"."SND"."UND"
```

c) The result of <specific type method> is *RV*.

13) The result of <binary value function> is the result of the simply contained <binary substring function>, <binary trim function>, or <binary overlay function>.

14) If <binary substring function> is specified, then

a) Let *B* be the value of the <binary value expression>, let *LB* be the length in octets of *B*, and let *S* be the value of the <start position>.

b) If <string length> is specified, then let *L* be the value of <string length> and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LB*+1 and *S*.

c) If either *B*, *S*, or *L* is the null value, then the result of the <binary substring function> is the null value.

d) If *E* is less than *S*, then an exception condition is raised: *data exception — substring error*.

e) Case:

i) If *S* is greater than *LB* or if *E* is less than 1 (one), then the result of the <binary substring function> is a zero-length string.

ii) Otherwise:

1) Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LB*+1. Let *L1* be *E1*–*S1*.

2) The result of the <binary substring function> is a binary string containing *L1* octets of *B* starting at octet number *S1* in the same order that the octets appear in *B*.

15) If <binary trim function> is specified, then

a) Let *S* be the value of the <trim source>.

b) Let *SO* be the value of <trim octet>.

c) If either *S* or *SO* the null value, then the result of the <binary trim function> is the null value.

d) If the length in octets of *SO* is not 1 (one), then an exception condition is raised: *data exception — trim error*.

e) Case:

i) If BOTH is specified or if no <trim specification> is specified, then the result of the <binary trim function> is the value of *S* with any leading or trailing octets equal to *SO* removed.

ii) If TRAILING is specified, then the result of the <binary trim function> is the value of *S* with any trailing octets equal to *SO* removed.

iii) If LEADING is specified, then the result of the <binary trim function> is the value of *S* with any leading octets equal to *SO* removed.

16) If the result of <string value expression> is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string*.

Conformance Rules

1) Without Feature T581, “Regular expression substring function”, conforming SQL language shall not contain a <regular expression substring function>.

2) Without Feature T312, “OVERLAY function”, conforming SQL language shall not contain a <character overlay function>.

3) Without Feature T312, “OVERLAY function”, conforming SQL language shall not contain a <binary overlay function>.

4) Without Feature T042, “Extended LOB data type support”, or Feature T022, “Advanced BINARY and VARBINARY data type support”, conforming SQL language shall not contain a <binary value function>.

5) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <character transliteration>.

6) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <transcoding>.

7) Without Feature T061, “UCS support”, conforming SQL language shall not contain a <normalize function>.

8) Without Feature S261, “Specific type method”, conforming SQL language shall not contain a <specific type method>.

9) Without Feature F394, “Optional normal form specification”, conforming SQL language shall not contain <normal form>.

10) Without Feature F844, “SUBSTRING_REGEXP”, conforming SQL language shall not contain <regex substring function>.

11) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language, <regex substring function> shall not contain <char length units>.

12) Without Feature F845, “TRANSLATE_REGEX”, conforming SQL language shall not contain <regex transliteration>.

13) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language, <regex transliteration> shall not contain <char length units>.

14) Without Feature F847, “Nonconstant regular expressions”, in conforming SQL language, <XQuery pattern>, <XQuery option flag>, and <XQuery replacement string> shall be <value specification>s.

6.30 <datetime value expression>

Function

Specify a datetime value.

Format

```
<datetime value expression> ::=
    <datetime term>
  | <interval value expression> <plus sign> <datetime term>
  | <datetime value expression> <plus sign> <interval term>
  | <datetime value expression> <minus sign> <interval term>

<datetime term> ::=
  <datetime factor>

<datetime factor> ::=
  <datetime primary> [ <time zone> ]

<datetime primary> ::=
    <value expression primary>
  | <datetime value function>

<time zone> ::=
  AT <time zone specifier>

<time zone specifier> ::=
    LOCAL
  | TIME ZONE <interval primary>
```

Syntax Rules

1) The declared type of a <datetime primary> shall be datetime.

2) If the <datetime value expression> immediately contains neither <plus sign> nor <minus sign>, then the precision of the result of the <datetime value expression> is the precision of the <value expression primary> or <datetime value function> that it simply contains.

3) If the declared type of the <datetime primary> is DATE, then <time zone> shall not be specified.

4) Case:

   a) If <time zone> is specified and the declared type of <datetime primary> is TIMESTAMP WITHOUT TIME ZONE or TIME WITHOUT TIME ZONE, then the declared type of <datetime term> is TIMESTAMP WITH TIME ZONE or TIME WITH TIME ZONE, respectively, with the same fractional seconds precision as <datetime primary>.

   b) Otherwise, the declared type of <datetime term> is the same as the declared type of <datetime primary>.

5) If the <datetime value expression> immediately contains either <plus sign> or <minus sign>, then:

   a) The <interval value expression> or <interval term> shall contain only <primary datetime field>s that are contained within the <datetime value expression> or <datetime term>.

b) The result of the <datetime value expression> contains the same <primary datetime field>s that are contained in the <datetime value expression> or <datetime term>, with a fractional seconds precision that is the greater of the fractional seconds precisions, if any, of either the <datetime value expression> and <interval term>, or the <datetime term> and <interval value expression> that it simply contains.

6) The declared type of the <interval primary> immediately contained in a <time zone specifier> shall be INTERVAL HOUR TO MINUTE.

Access Rules

None.

General Rules

1) If the value of any <datetime primary>, <interval value expression>, <datetime value expression>, or <interval term> simply contained in a <datetime value expression> is the null value, then the result of the <datetime value expression> is the null value.

2) If <time zone> is specified and the <interval primary> immediately contained in <time zone specifier> is the null value, then the result of the <datetime value expression> is the null value.

3) The value of a <datetime primary> is the value of the immediately contained <value expression primary> or <datetime value function>.

4) In the following General Rules, arithmetic is performed so as to maintain the integrity of the datetime data type that is the result of the <datetime term> or <datetime value expression>. This may involve carry from or to the immediately next more significant <primary datetime field>. If the data type of the <datetime term> or <datetime value expression> is time with or without time zone, then arithmetic on the HOUR <primary datetime field> is undertaken modulo 24. If the <interval value expression> or <interval term> is a year-month interval, then the DAY field of the result is the same as the DAY field of the <datetime term> or <datetime value expression>.

5) The value of a <datetime term> is determined as follows. Let *DT* be the declared type, *DV* the UTC component of the value, and *TZD* the time zone component, if any, of the <datetime primary> simply contained in the <datetime term>, and let *STZD* be the current default time zone displacement of the SQL-session.

   Case:

   a) If <time zone> is not specified, then the value of <datetime term> is *DV*.

   b) Otherwise:

      i) Case:

         1) If *DT* is datetime with time zone, then the UTC component of the <datetime term> is *DV*.

         2) Otherwise, the UTC component of the <datetime term> is *DV* – *STZD*.

      ii) Case:

         1) If LOCAL is specified, then let *TZ* be *STZD*.

         2) If TIME ZONE is specified, then, if the value of the <interval primary> immediately contained in <time zone specifier> is less than `INTERVAL -'12:59'` or greater than

`INTERVAL +'14:00'`, then an exception condition is raised: *data exception — invalid time zone displacement value*. Otherwise, let *TZ* be the value of the <interval primary> simply contained in <time zone>.

iii) The time zone component of the value of the <datetime term> is *TZ*.

6) If a <datetime value expression> immediately contains the operator <plus sign> or <minus sign>, then the time zone component, if any, of the result is the same as the time zone component of the immediately contained <datetime term> or <datetime value expression>. The result (if the result type is without time zone) or the UTC component of the result (if the result type has time zone) is effectively evaluated as follows:

a) Case:

i) If <datetime value expression> immediately contains the operator <plus sign> and the <interval value expression> or <interval term> is not negative, or if <datetime value expression> immediately contains the operator <minus sign> and the <interval term> is negative, then successive <primary datetime field>s of the <interval value expression> or <interval term> are added to the corresponding fields of the <datetime value expression> or <datetime term>.

ii) Otherwise, successive <primary datetime field>s of the <interval value expression> or <interval term> are subtracted from the corresponding fields of the <datetime value expression> or <datetime term>.

b) If, after the preceding step, any <primary datetime field> of the result is outside the permissible range of values for the field or the result is invalid based on the natural rules for dates and times, then an exception condition is raised: *data exception — datetime field overflow*.

NOTE 129 — For the permissible range of values for <primary datetime field>s, see Table 9, “Valid values for datetime fields”.

Conformance Rules

1) Without Feature F052, “Intervals and datetime arithmetic”, conforming SQL language shall not contain <datetime value expression> that immediately contains a <plus sign> or a <minus sign>.

2) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <time zone>.

6.31 <datetime value function>

Function

Specify a function yielding a value of type datetime.

Format

```
<datetime value function> ::=
    <current date value function>
  | <current time value function>
  | <current timestamp value function>
  | <current local time value function>
  | <current local timestamp value function>

<current date value function> ::=
  CURRENT_DATE

<current time value function> ::=
  CURRENT_TIME [ <left paren> <time precision> <right paren> ]

<current local time value function> ::=
  LOCALTIME [ <left paren> <time precision> <right paren> ]

<current timestamp value function> ::=
  CURRENT_TIMESTAMP [ <left paren> <timestamp precision> <right paren> ]

<current local timestamp value function> ::=
  LOCALTIMESTAMP [ <left paren> <timestamp precision> <right paren> ]
```

Syntax Rules

1) The declared type of a <current date value function> is DATE. The declared type of a <current time value function> is TIME WITH TIME ZONE. The declared type of a <current timestamp value function> is TIMESTAMP WITH TIME ZONE.

   NOTE 130 — See the Syntax Rules of Subclause 6.1, "<data type>", for rules governing <time precision> and <timestamp precision>.

2) If <time precision> *TP* is specified, then LOCALTIME(*TP*) is equivalent to:

   ```
   CAST (CURRENT_TIME(TP) AS
   TIME(TP) WITHOUT TIME
   ZONE)
   ```

   Otherwise, LOCALTIME is equivalent to:

   ```
   CAST (CURRENT_TIME AS TIME WITHOUT
   TIME ZONE)
   ```

3) If <timestamp precision> *TP* is specified, then LOCALTIMESTAMP(*TP*) is equivalent to:

   ```
   CAST (CURRENT_TIMESTAMP(TP) AS
   TIMESTAMP(TP) WITHOUT TIME
   ZONE)
   ```

Otherwise, LOCALTIMESTAMP is equivalent to:

```
CAST (CURRENT_TIMESTAMP AS TIMESTAMP
WITHOUT TIME ZONE)
```

Access Rules

None.

General Rules

1) The <datetime value function>s CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP respectively return the current date, current time, and current timestamp; the time and timestamp values are returned with time zone displacement equal to the current default time zone displacement of the SQL-session.

2) If specified, <time precision> and <timestamp precision> respectively determine the precision of the time or timestamp value returned.

3) Let *S* be an <SQL procedure statement> that is not generally contained in a <triggered action>. All <datetime value function>s that are contained in <value expression>s that are generally contained, without an intervening <routine invocation> whose subject routines do not include an SQL function, either in *S* without an intervening <SQL procedure statement> or in an <SQL procedure statement> contained in the <triggered action> of a trigger activated as a consequence of executing *S*, are effectively evaluated simultaneously. The time of evaluation of a <datetime value function> during the execution of *S* and its activated triggers is implementation-dependent.

   NOTE 131 — Activation of triggers is defined in Subclause 4.38.2, “Trigger execution”.

Conformance Rules

1) Without Feature F555, “Enhanced seconds precision”, conforming SQL language shall not contain a <current local time value function> that contains a <time precision> that is not 0 (zero).

2) Without Feature F555, “Enhanced seconds precision”, conforming SQL language shall not contain a <current local timestamp value function> that contains a <timestamp precision> that is neither 0 (zero) nor 6.

3) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <current time value function>.

4) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <current timestamp value function>.

6.32 <interval value expression>

Function

Specify an interval value.

Format

```
<interval value expression> ::=
    <interval term>
  | <interval value expression 1> <plus sign> <interval term 1>
  | <interval value expression 1> <minus sign> <interval term 1>
  | <left paren> <datetime value expression> <minus sign> <datetime term> <right paren>
      <interval qualifier>

<interval term> ::=
    <interval factor>
  | <interval term 2> <asterisk> <factor>
  | <interval term 2> <solidus> <factor>
  | <term> <asterisk> <interval factor>

<interval factor> ::=
  [ <sign> ] <interval primary>

<interval primary> ::=
    <value expression primary> [ <interval qualifier> ]
  | <interval value function>

<interval value expression 1> ::=
  <interval value expression>

<interval term 1> ::=
  <interval term>

<interval term 2> ::=
  <interval term>
```

Syntax Rules

1) The declared type of an <interval value expression> is interval. The declared type of a <value expression primary> immediately contained in an <interval primary> shall be interval.

2) Case:

   a) If the <interval value expression> simply contains an <interval qualifier> *IQ*, then the declared type of the result is INTERVAL *IQ*.

   b) If the <interval value expression> is an <interval term>, then the result of the <interval value expression> contains the same interval fields as the <interval primary>. If the <interval primary> contains a seconds field, then the result's fractional seconds precision is the same as the <interval primary>'s fractional seconds precision. The result's <interval leading field precision> is implementation-defined, but shall not be less than the <interval leading field precision> of the <interval primary>.

c) If <interval term 1> is specified, then the result contains every interval field that is contained in the result of either <interval value expression 1> or <interval term 1>, and, if both contain a seconds field, then the fractional seconds precision of the result is the greater of the two fractional seconds precisions. The <interval leading field precision> is implementation-defined, but shall be sufficient to represent all interval values with the interval fields and <interval leading field precision> of <interval value expression 1> as well as all interval values with the interval fields and <interval leading field precision> of <interval term 1>.

NOTE 132 — Interval fields are effectively defined by Table 4, "Fields in year-month INTERVAL values", and Table 5, "Fields in day-time INTERVAL values".

3) Case:

a) If <interval term 1> is a year-month interval, then <interval value expression 1> shall be a year-month interval.

b) If <interval term 1> is a day-time interval, then <interval value expression 1> shall be a day-time interval.

4) If <datetime value expression> is specified, then <datetime value expression> and <datetime term> shall be comparable.

5) An <interval primary> shall specify <interval qualifier> only if the <interval primary> specifies a <dynamic parameter specification>.

Access Rules

None.

General Rules

1) If an <interval term> specifies "<term> * <interval factor>", then let *T* and *F* be respectively the value of the <term> and the value of the <interval factor>. The result of the <interval term> is the result of *F* * *T*.

2) If the value of any <interval primary>, <datetime value expression>, <datetime term>, or <factor> that is simply contained in an <interval value expression> is the null value, then the result of the <interval value expression> is the null value.

3) If *IP* is an <interval primary>, then

Case:

a) If *IP* immediately contains a <value expression primary> *VEP* and an explicit <interval qualifier> *IQ*, then the value of *IP* is computed by:

```
CAST ( VEP AS INTERVAL IQ )
```

b) If *IP* immediately contains a <value expression primary> *VEP*, then the value of *IP* is the value of *VEP*.

c) If *IP* is an <interval value function> *IVF*, then the value of *IP* is the value of *IVF*.

4) If the <sign> of an <interval factor> is <minus sign>, then the value of the <interval factor> is the negative of the value of the <interval primary>; otherwise, the value of an <interval factor> is the value of the <interval primary>.

5) If <interval term 2> is specified, then:

   a) Let *X* be the value of <interval term 2> and let *Y* be the value of <factor>.

   b) Let *P* and *Q* be respectively the most significant and least significant <primary datetime field>s of <interval term 2>.

   c) Let *E* be an exact numeric result of the operation

      ```
      CAST ( CAST ( X AS INTERVAL Q )
      AS E1 )
      ```

      where *E1* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

   d) Let *OP* be the operator * or / specified in the <interval value expression>.

   e) Let *I*, the result of the <interval value expression> expressed in terms of the <primary datetime field> *Q*, be the result of

      ```
      CAST ( (E OP Y) AS
      INTERVAL Q )
      ```

   f) The result of the <interval value expression> is

      ```
      CAST ( I AS INTERVAL W )
      ```

      where *W* is an <interval qualifier> identifying the <primary datetime field>s *P* TO *Q*, but with <interval leading field precision> such that significant digits are not lost.

6) If <interval term 1> is specified, then let *P* and *Q* be respectively the most significant and least significant <primary datetime field>s in <interval term 1> and <interval value expression 1>, let *X* be the value of <interval value expression 1>, and let *Y* be the value of <interval term 1>.

   a) Let *A* be an exact numeric result of the operation

      ```
      CAST ( CAST ( X AS INTERVAL Q )
      AS E1 )
      ```

      where *E1* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

   b) Let *B* be an exact numeric result of the operation

      ```
      CAST ( CAST ( Y AS INTERVAL Q )
      AS E2 )
      ```

      where *E2* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

   c) Let *OP* be the operator + or – specified in the <interval value expression>.

   d) Let *I*, the result of the <interval value expression> expressed in terms of the <primary datetime field> *Q*, be the result of:

      ```
      CAST ( (A OP B) AS
      INTERVAL Q )
      ```

e) The result of the <interval value expression> is

```
CAST ( I AS INTERVAL W )
```

where *W* is an <interval qualifier> identifying the <primary datetime field>s *P* TO *Q*, but with <interval leading field precision> such that significant digits are not lost.

7) If <datetime value expression> is specified, then let *Y* be the least significant <primary datetime field> specified by <interval qualifier>. Let *DTE* be the <datetime value expression>, let *DT* be the <datetime term>, and let *MSP* be the implementation-defined maximum seconds precision. Evaluation of <interval value expression> proceeds as follows:

a) Case:

i) If the declared type of <datetime value expression> is TIME WITH TIME ZONE, then let *A* be the value of:

```
CAST ( DTE AT LOCAL
AS TIME(MSP) WITHOUT
TIME ZONE )
```

ii) If the declared type of <datetime value expression> is TIMESTAMP WITH TIME ZONE, then let *A* be the value of:

```
CAST ( DTE AT LOCAL
AS TIMESTAMP(MSP) WITHOUT
TIME ZONE )
```

iii) Otherwise, let *A* be the value of *DTE*.

b) Case:

i) If the declared type of <datetime term> is TIME WITH TIME ZONE, then let *B* be the value of:

```
CAST ( DT AT LOCAL
AS TIME(MSP) WITHOUT
TIME ZONE )
```

ii) If the declared type of <datetime term> is TIMESTAMP WITH TIME ZONE, then let *B* be the value of:

```
CAST ( DT AT LOCAL
AS TIMESTAMP(MSP) WITHOUT
TIME ZONE )
```

iii) Otherwise, let *B* be the value of *DTE*.

c) *A* and *B* are converted to integer scalars *A2* and *B2* respectively in units *Y* as displacements from some implementation-dependent start datetime.

d) The result is determined by effectively computing *A2–B2* and then converting the difference to an interval using an <interval qualifier> whose <end field> is *Y* and whose <start field> is sufficiently significant to avoid loss of significant digits. The difference of two values of type TIME (with or without time zone) is constrained to be between –24:00:00 and +24:00:00 (excluding each end point); it is implementation-defined which of two non-zero values in this range is the result, although the

computation shall be deterministic. That interval is then converted to an interval using the specified <interval qualifier>, rounding or truncating if necessary. The choice of whether to round or truncate is implementation-defined. If the required number of significant digits exceeds the implementation-defined maximum number of significant digits, then an exception condition is raised: *data exception — interval field overflow*.

Conformance Rules

1) Without Feature F052, “Intervals and datetime arithmetic”, conforming SQL language shall not contain an <interval value expression>.

6.33 <interval value function>

Function

Specify a function yielding a value of type interval.

Format

```
<interval value function> ::=
  <interval absolute value function>

<interval absolute value function> ::=
  ABS <left paren> <interval value expression> <right paren>
```

Syntax Rules

1) If <interval absolute value function> is specified, then the declared type of the result is the declared type of the <interval value expression>.

Access Rules

None.

General Rules

1) If <interval absolute value function> is specified, then let *N* be the value of the <interval value expression>.

   Case:

   a) If *N* is the null value, then the result is the null value.

   b) If $N \geq 0$ (zero), then the result is *N*.

   c) Otherwise, the result is $-1 * N$.

Conformance Rules

1) Without Feature F052, “Intervals and datetime arithmetic”, conforming SQL shall not contain an <interval value function>.

6.34 <boolean value expression>

Function

Specify a boolean value.

Format

```
<boolean value expression> ::=
    <boolean term>
  | <boolean value expression> OR <boolean term>

<boolean term> ::=
    <boolean factor>
  | <boolean term> AND <boolean factor>

<boolean factor> ::=
  [ NOT ] <boolean test>

<boolean test> ::=
  <boolean primary> [ IS [ NOT ] <truth value> ]

<truth value> ::=
    TRUE
  | FALSE
  | UNKNOWN

<boolean primary> ::=
    <predicate>
  | <boolean predicand>

<boolean predicand> ::=
    <parenthesized boolean value expression>
  | <nonparenthesized value expression primary>

<parenthesized boolean value expression> ::=
  <left paren> <boolean value expression> <right paren>
```

Syntax Rules

1) The declared type of a <nonparenthesized value expression primary> shall be boolean.

2) If NOT is specified in a <boolean test>, then let *BP* be the contained <boolean primary> and let *TV* be the contained <truth value>. The <boolean test> is equivalent to:

   (NOT (*BP* IS *TV*))

3) Let *X* denote either a column *C* or the <key word> VALUE. Given a <boolean value expression> *BVE* and *X*, the notion "*BVE* is a *known-not-null condition* for *X*" is defined recursively as follows:

   a) If *BVE* is a <predicate>, then

      Case:

i) If *BVE* is a <predicate> of the form "*RVE* IS NOT NULL", where *RVE* is a <row value predicand> that is a <row value constructor predicand> that simply contains a <common value expression>, <boolean predicand>, or <row value constructor element> that is a <column reference> that references *C*, then *BVE* is a known-not-null condition for *C*.

ii) If *BVE* is the <predicate> "VALUE IS NOT NULL", then *BVE* is a known-not-null condition for VALUE.

iii) Otherwise, *BVE* is not a known-not-null condition for *X*.

b) If *BVE* is a <parenthesized boolean value expression> and the simply contained <boolean value expression> is a known-not-null condition for *X*, then *BVE* is a known-not-null condition for *X*.

c) If *BVE* is a <nonparenthesized value expression primary>, then *BVE* is not a known-not-null condition for *X*.

d) If *BVE* is a <boolean test>, then let *BP* be the <boolean primary> immediately contained in *BVE*. If *BP* is a known-not-null condition for *X*, and <truth value> is not specified, then *BVE* is a known-not-null condition for *X*. Otherwise, *BVE* is not a known-not-null condition for *X*.

e) If *BVE* is of the form "NOT *BT*", where *BT* is a <boolean test>, then

Case:

i) If *BT* is "*CR* IS NULL", where *CR* is a column reference that references column *C*, then *BVE* is a known-not-null condition for *C*.

ii) If *BT* is "VALUE IS NULL", then *BVE* is a known-not-null condition for VALUE.

iii) Otherwise, *BVE* is not a known-not-null condition for *X*.

NOTE 133 — For simplicity, this rule does not attempt to analyze conditions such as "NOT NOT *A* IS NULL", or "NOT (*A* IS NULL OR NOT (*B* = 2))"

f) If *BVE* is of the form "*BVE1* AND *BVE2*", then

Case:

i) If either *BVE1* or *BVE2* is a known-not-null condition for *X*, then *BVE* is aknown-not-null condition for *X*.

ii) Otherwise, *BVE* is not a known-not-null condition for *X*.

g) If *BVE* is of the form "*BVE1* OR *BVE2*", then *BVE* is not a known-not-null condition for *X*.

NOTE 134 — For simplicity, this rule does not detect cases such as "*A* IS NOT NULL OR *A* IS NOT NULL", which might be classified as a known-not-null condition.

4) The notion of "retrospectively deterministic" is defined recursively as follows:

a) A <parenthesized boolean value expression> is retrospectively deterministic if the simply contained <boolean value expression> is retrospectively deterministic.

b) A <nonparenthesized value expression primary> is retrospectively deterministic if it is not possibly non-deterministic.

c) A <predicate> *P* is *retrospectively deterministic* if one of the following is true:

i) *P* is not possibly non-deterministic.

ii) *P* is a <comparison predicate> of the form "*X* < *Y*", "*X* <= *Y*", "*Y* > *X*", "*Y* >= *X*", "*X* < *Y* + *Z*", "*X* <= *Y* + *Z*", "*Y* + *Z* > *X*", "*Y* + *Z* >= *X*", "*X* < *Y* – *Z*", "*X* <= *Y* – *Z*", "*Y* – *Z* > *X*", or "*Y* – *Z* >= *X*", where *Y* is CURRENT_DATE, CURRENT_TIMESTAMP or LOCALTIMESTAMP, *X* and *Z* are not possibly non-deterministic <value expression>s, and the declared types of the left and right comparands are either both datetime with time zone or both datetime without time zone.

iii) *P* is a <quantified comparison predicate> of the form "*Y* > <quantifier> <table subquery>", "*Y* + *Z* > <quantifier> <table subquery>", "*Y* – *Z* > <quantifier> <table subquery>", "*Y* >= <quantifier> <table subquery>", "*Y* + *Z* >= <quantifier> <table subquery>", or "*Y* - *Z* >= <quantifier> <table subquery>", where *Y* is CURRENT_DATE, CURRENT_TIMESTAMP or LOCALTIMESTAMP, *Z* is a <value expression> that is not possibly non-deterministic, the <query expression> simply contained in the <table subquery> is not possibly non-deterministic, and the declared types of the left and right comparands are either both datetime with time zone or both datetime without time zone.

iv) *P* is a <between predicate> that is transformed into a retrospectively deterministic <boolean value expression>.

d) A <boolean primary> is retrospectively deterministic if the simply contained <predicate>, <parenthesized boolean value expression> or <nonparenthesized value expression primary> is retrospectively deterministic.

e) Let *BF* be a <boolean factor>. Let *BP* be the <boolean primary> simply contained in *BF*.

i) *BF* is called *negative* if *BF* is of any of the following forms:

```
NOT BP
BP IS FALSE
BP IS NOT TRUE
NOT BP IS NOT FALSE
NOT BP IS TRUE
```

ii) *BF* is retrospectively deterministic if one of the following is true:

1) *BF* is negative and *BF* does not generally contain a possibly nondeterministic <value expression>.

2) *BF* is not negative and *BP* is retrospectively deterministic.

f) A <boolean value expression> is retrospectively deterministic if every simply contained <boolean factor> is retrospectively deterministic.

Access Rules

None.

General Rules

1) The result is derived by the application of the specified boolean operators ("AND", "OR", "NOT", and "IS") to the results derived from each <boolean primary>. If boolean operators are not specified, then the result of the <boolean value expression> is the result of the specified <boolean primary>.

2) NOT (*True*) is *False*, NOT (*False*) is *True*, and NOT (*Unknown*) is *Unknown*.

3) Table 11, “Truth table for the AND boolean operator”, Table 12, “Truth table for the OR boolean operator”, and Table 13, “Truth table for the IS boolean operator” specify the semantics of AND, OR, and IS, respectively.

Table 11 — Truth table for the AND boolean operator

AND	***True***	***False***	***Unknown***
True	*True*	*False*	*Unknown*
False	*False*	*False*	*False*
Unknown	*Unknown*	*False*	*Unknown*

Table 12 — Truth table for the OR boolean operator

OR	***True***	***False***	***Unknown***
True	*True*	*True*	*True*
False	*True*	*False*	*Unknown*
Unknown	*True*	*Unknown*	*Unknown*

Table 13 — Truth table for the IS boolean operator

IS	**TRUE**	**FALSE**	**UNKNOWN**
True	*True*	*False*	*False*
False	*False*	*True*	*False*
Unknown	*False*	*False*	*True*

Conformance Rules

1) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <boolean primary> that simply contains a <nonparenthesized value expression primary>.

2) Without Feature F571, “Truth value tests”, conforming SQL language shall not contain a <boolean test> that simply contains a <truth value>.

6.35 <array value expression>

Function

Specify an array value.

Format

```
<array value expression> ::=
    <array concatenation>
  | <array primary>

<array concatenation> ::=
  <array value expression 1> <concatenation operator> <array primary>

<array value expression 1> ::=
  <array value expression>

<array primary> ::=
  <value expression primary>
```

Syntax Rules

1) The declared type of the <array value expression> is the declared type of the immediately contained <array concatenation> or <array primary>.

2) The declared type of <array primary> is the declared type of the immediately contained <value expression primary>, which shall be an array type.

3) If <array concatenation> is specified, then:

   a) Let *DT* be the data type determined by applying Subclause 9.3, "Result of data type combinations", to the declared types of <array value expression 1> and <array primary>.

   b) Let *IMDC* be the implementation-defined maximum cardinality of an array type.

   c) The declared type of the result of <array concatenation> is an array type whose element type is the element type of *DT* and whose maximum cardinality is the lesser of *IMDC* and the sum of the maximum cardinality of <array value expression 1> and the maximum cardinality of <array primary>.

Access Rules

None.

General Rules

1) The value of the result of <array value expression> is the value of the immediately contained <array concatenation> or <array primary>.

2) If <array concatenation> is specified, then let *AV1* be the value of <array value expression 1> and let *AV2* be the value of <array primary>.

Case:

a) If either *AV1* or *AV2* is the null value, then the result of the <array concatenation> is the null value.

b) If the sum of the cardinality of *AV1* and the cardinality of *AV2* is greater than *IMDC*, then an exception condition is raised: *data exception — array data, right truncation*.

c) Otherwise, the result is the array comprising every element of *AV1* followed by every element of *AV2*.

Conformance Rules

1) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array value expression>.

6.36 <array value constructor>

Function

Specify construction of an array.

Format

```
<array value constructor> ::=
    <array value constructor by enumeration>
  | <array value constructor by query>

<array value constructor by enumeration> ::=
  ARRAY <left bracket or trigraph> <array element list> <right bracket or trigraph>

<array element list> ::=
  <array element> [ { <comma> <array element> }... ]

<array element> ::=
  <value expression>

<array value constructor by query> ::=
  ARRAY <left paren> <query expression> [ <order by clause> ] <right paren>
```

Syntax Rules

1) The declared type of <array value constructor> is the declared type of the immediately contained <array value constructor by enumeration> or <array value constructor by query>.

2) The declared type of the <array value constructor by enumeration> is an array type with element type *DT*, where *DT* is the declared type determined by applying Subclause 9.3, "Result of data type combinations", to the declared types of the <array element>s immediately contained in the <array element list> of this <array value constructor by enumeration>. The maximum cardinality is the number of <array element>s in the <array element list>, which shall not be greater than the implementation-defined maximum cardinality for array types whose element type is *DT*.

3) If <array value constructor by query> is specified, then

   a) The <query expression> shall be of degree 1 (one). Let *ET* be the declared type of the column in the result of <query expression>.

   b) The declared type of the <array value constructor by query> is array with element type *ET* and maximum cardinality equal to the implementation-defined maximum cardinality *IMDC* for such array types.

Access Rules

None.

General Rules

1) The value of <array value constructor> is the value of the immediately contained <array value constructor by enumeration> or <array value constructor by query>.

2) The result of <array value constructor by enumeration> is an array whose *i*-th element is the value of the *i*-th <array element> immediately contained in the <array element list>, cast as the data type of *DT*.

3) The result of <array value constructor by query> is determined as follows:

   a) The <query expression> is evaluated, producing a table *T*. Let *N* be the number of rows in *T*.

   b) If *N* is greater than *IMDC*, then an exception condition is raised: *data exception — array data, right truncation*.

   c) *T* is ordered according to the <sort specification list>. If there is no <sort specification list>, then the ordering is implementation-dependent.

   d) The result of <array value constructor by query> is an array of *N* elements such that for all *i*, 1 (one) $\leq i \leq N$, the value of the *i*-th element is the value of the only column in the *i*-th row of *T*, as ordered by GR 3)c)the .

Conformance Rules

1) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array value constructor by enumeration>.

2) Without Feature S095, “Array constructors by query”, conforming SQL language shall not contain an <array value constructor by query>.

6.37 <multiset value expression>

Function

Specify a multiset value.

Format

```
<multiset value expression> ::=
    <multiset term>
  | <multiset value expression> MULTISET UNION [ ALL | DISTINCT ] <multiset term>
  | <multiset value expression> MULTISET EXCEPT [ ALL | DISTINCT ] <multiset term>

<multiset term> ::=
    <multiset primary>
  | <multiset term> MULTISET INTERSECT [ ALL | DISTINCT ] <multiset primary>

<multiset primary> ::=
    <multiset value function>
  | <value expression primary>
```

Syntax Rules

1) The declared type of a <multiset primary> is the declared type of the immediately contained <multiset value function> or <value expression primary>, which shall be a multiset type.

2) If *MI* is a <multiset term> that immediately contains MULTISET INTERSECT, then let *OP1* be the first operand (the <multiset term>) and let *OP2* be the second operand (the <multiset primary>).

   a) *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

   b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. Let *ET* be the data type determined by Subclause 9.3, "Result of data type combinations", using the types *ET1* and *ET2*. The result type of the MULTISET INTERSECT operation is multiset with element type *ET*.

   c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

   d) *MI* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                       FROM UNNEST (OP1) AS T1(V)
                     INTERSECT SQ
                       SELECT T2.V
                       FROM UNNEST (OP2) AS T2(V)
                     )
  END )
```

3) If *MU* is a <multiset value expression> that immediately contains MULTISET UNION, then let *OP1* be the first operand (the <multiset value expression>) and let *OP2* be the second operand (the <multiset term>).

a) If DISTINCT is specified, then *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. Let *ET* be the data type determined by Subclause 9.3, "Result of data type combinations", using the types *ET1* and *ET2*. The result type of the MULTISET UNION operation is multiset with element type *ET*.

c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

d) *MU* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       UNION SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

4) If *ME* is a <multiset value expression> that immediately contains MULTISET EXCEPT, then let *OP1* be the first operand (the <multiset term>) and let *OP2* be the second operand (the <multiset primary>).

a) *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. Let *ET* be the data type determined by Subclause 9.3, "Result of data type combinations", using the types *ET1* and *ET2*. The result type of the MULTISET EXCEPT operation is multiset with element type *ET*.

c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

d) *ME* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       EXCEPT SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

Access Rules

None.

General Rules

1) The value of a <multiset primary> is the value of the immediately contained <multiset value function> or <value expression primary>.

2) The value of a <multiset term> that is a <multiset primary> is the value of the <multiset primary>.

3) The value of a <multiset value expression> that is a <multiset term> is the value of <multiset term>.

Conformance Rules

1) Without Feature S275, “Advanced multiset support”, conforming SQL language shall not contain MULTISET UNION, MULTISET INTERSECTION, or MULTISET EXCEPT.

NOTE 135 — If MULTISET UNION DISTINCT, MULTISET INTERSECTION, or MULTISET EXCEPT is specified, then the Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

6.38 <multiset value function>

Function

Specify a function yielding a value of a multiset type.

Format

```
<multiset value function> ::=
  <multiset set function>

<multiset set function> ::=
  SET <left paren> <multiset value expression> <right paren>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression> simply contained in <multiset set function>. *MVE* is a multiset operand of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

2) The <multiset set function> is equivalent to

```
( CASE WHEN MVE IS NULL THEN NULL
       ELSE MULTISET ( SELECT DISTINCT M.E
                       FROM UNNEST (MVE) AS M(E) )
  END )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset value function>.

NOTE 136 — The Conformance Rules of Subclause 9.11, "Multiset element grouping operations", also apply.

6.39 <multiset value constructor>

Function

Specify construction of a multiset.

Format

```
<multiset value constructor> ::=
    <multiset value constructor by enumeration>
  | <multiset value constructor by query>
  | <table value constructor by query>

<multiset value constructor by enumeration> ::=
  MULTISET <left bracket or trigraph> <multiset element list> <right bracket or trigraph>

<multiset element list> ::=
  <multiset element> [ { <comma> <multiset element> }... ]

<multiset element> ::=
  <value expression>

<multiset value constructor by query> ::=
  MULTISET <left paren> <query expression> <right paren>

<table value constructor by query> ::=
  TABLE <left paren> <query expression> <right paren>
```

Syntax Rules

1) If <multiset value constructor> immediately contains a <table value constructor by query> *TVCBQ*, then:

   a) Let *QE* be the <query expression> simply contained in *TVCBQ*.

   b) Let *n* be the number of columns in the result of *QE*.

   c) Let C_1, ..., C_n be implementation-dependent identifiers that are all distinct from one another.

   d) *TVCBQ* is equivalent to

   ```
   MULTISET ( SELECT ROW ( C1, ..., Cn )
   FROM ( QE ) AS T ( C1, ..., Cn ) )
   ```

2) The declared type of <multiset value constructor> is the declared type of the immediately contained <multiset value constructor by enumeration> or <multiset value constructor by query>.

3) The declared type of the <multiset value constructor by enumeration> is a multiset type with element type *DT*, where *DT* is the declared type determined by applying Subclause 9.3, "Result of data type combinations", to the declared types of the <multiset element>s immediately contained in the <multiset element list> of this <multiset value constructor by enumeration>.

4) If <multiset value constructor by query> is specified, then

a) The <query expression> shall be of degree 1 (one). Let *ET* be the declared type of the column in the result of <query expression>.

b) The declared type of the <multiset value constructor by query> is multiset with element type *ET*.

Access Rules

None.

General Rules

1) The value of <multiset value constructor> is the value of the immediately contained <multiset value constructor by enumeration> or <multiset value constructor by query>.

2) The result of <multiset value constructor by enumeration> is a multiset whose elements are the values of the <multiset element>s immediately contained in the <multiset element list>, cast as the data type of *DT*.

3) If <multiset value constructor by query> is specified, then:

   a) The <query expression> is evaluated, producing a table *T*. Let *N* be the number of rows in *T*.

   b) The result of <multiset value constructor by query> is a multiset of *N* elements, with one element for each row of *T*, where the value of each element is the value of the only column in the corresponding row of *T*.

Conformance Rules

1) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <multiset value constructor>.

2) Without Feature T326, “Table functions”, a <multiset value constructor> shall not contain a <table value constructor by query>.

(Blank page)

7 Query expressions

This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-4.
This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-9.
This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-14.

7.1 <row value constructor>

Function

Specify a value or list of values to be constructed into a row.

Format

```
<row value constructor> ::=
    <common value expression>
  | <boolean value expression>
  | <explicit row value constructor>

<explicit row value constructor> ::=
    <left paren> <row value constructor element> <comma>
        <row value constructor element list> <right paren>
  | ROW <left paren> <row value constructor element list> <right paren>
  | <row subquery>

<row value constructor element list> ::=
  <row value constructor element> [ { <comma> <row value constructor element> }... ]

<row value constructor element> ::=
  <value expression>

<contextually typed row value constructor> ::=
    <common value expression>
  | <boolean value expression>
  | <contextually typed value specification>
  | <left paren> <contextually typed value specification> <right paren>
  | <left paren> <contextually typed row value constructor element> <comma>
      <contextually typed row value constructor element list> <right paren>
  | ROW <left paren> <contextually typed row value constructor element list> <right paren>

<contextually typed row value constructor element list> ::=
  <contextually typed row value constructor element>
      [ { <comma> <contextually typed row value constructor element> }... ]

<contextually typed row value constructor element> ::=
    <value expression>
  | <contextually typed value specification>
```

```
<row value constructor predicand> ::=
    <common value expression>
  | <boolean predicand>
  | <explicit row value constructor>
```

Syntax Rules

1) If a <row value constructor> is a <common value expression> or a <boolean value expression> *X*, then the <row value constructor> is equivalent to

   `ROW ( X )`

2) If a <row value constructor predicand> is a <common value expression> or a <boolean predicand> *X*, then the <row value constructor predicand> is equivalent to

   `ROW ( X )`

3) Let *ERVC* be an <explicit row value constructor>.

   Case:

   a) If *ERVC* simply contains a <row subquery>, then the declared type of *ERVC* is the declared type of that <row subquery>.

   b) Otherwise, the declared type of *ERVC* is a row type described by a sequence of (<field name>, <data type>) pairs, corresponding in order to each <row value constructor element> *X* simply contained in *ERVC*. The data type is the declared type of *X* and the <field name> is implementation-dependent.

4) If a <row value constructor> or <row value constructor predicand> *RVC* is an <explicit row value constructor> *ERVC*, then the declared type of *RVC* is the declared type of *ERVC*.

5) Let *CTRVC* be the <contextually typed row value constructor>.

   a) If *CTRVC* is a <common value expression>, <boolean value expression>, or <contextually typed value specification> *X*, then *CTRVC* is equivalent to:

      `ROW ( X )`

   b) After the syntactic transformation specified in SR 5)a) has been performed, if necessary, the declared type of *CTRVC* is a row type described by a sequence of (<field name>, <data type>) pairs, corresponding in order to each <contextually typed row value constructor element> *X* simply contained in *CTRVC*. The <data type> is the declared type of *X* and the <field name> is implementation-dependent.

6) The degree of a <row value constructor>, <contextually typed row value constructor>, or <row value constructor predicand> is the degree of its declared type.

Access Rules

None.

General Rules

1) The value of a <null specification> is the null value.

2) The value of a <default specification> is determined according to the General Rules of Subclause 11.5, "<default clause>".

3) The value of an <empty specification> is an empty collection.

4) Case:

   a) If a <row value constructor>, <row value constructor predicand>, or <contextually typed row value constructor> immediately contains a <common value expression>, <boolean value expression>, or <contextually typed row value constructor element> *X*, then the result of the <row value constructor>, <row value constructor predicand>, or <contextually typed row value constructor> is a row containing a single column whose value is the value of *X*.

   b) If an <explicit row value constructor> is specified, then the result of the <row value constructor> or <row value constructor predicand> is a row of columns, the value of whose *i*-th column is the value of the *i*-th <row value constructor element> simply contained in the <explicit row value constructor>.

   c) If a <contextually typed row value constructor element list> is specified, then the result of the <contextually typed row value constructor> is a row of columns, the value of whose *i*-th column is the value of the *i*-th <contextually typed row value constructor element> in the <contextually typed row value constructor element list>.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain an <explicit row value constructor> that immediately contains ROW.

2) Without Feature T051, "Row types", conforming SQL language shall not contain a <contextually typed row value constructor> that immediately contains ROW.

3) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain an <explicit row value constructor> that is not simply contained in a <table value constructor> and that contains more than one <row value constructor element>.

4) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain an <explicit row value constructor> that is a <row subquery>.

5) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <row value constructor predicand> that immediately contains a <boolean predicand>.

6) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain a <contextually typed row value constructor> that is not simply contained in a <contextually typed table value constructor> and that contains more than one <row value constructor element>.

7) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain a <contextually typed row value constructor> that is a <row subquery>.

7.2 <row value expression>

Function

Specify a row value.

Format

```
<row value expression> ::=
    <row value special case>
  | <explicit row value constructor>

<table row value expression> ::=
    <row value special case>
  | <row value constructor>

<contextually typed row value expression> ::=
    <row value special case>
  | <contextually typed row value constructor>

<row value predicand> ::=
    <row value special case>
  | <row value constructor predicand>

<row value special case> ::=
  <nonparenthesized value expression primary>
```

Syntax Rules

1) The declared type of a <row value special case> shall be a row type.

2) The declared type of a <row value expression> is the declared type of the immediately contained <row value special case> or <explicit row value constructor>.

3) The declared type of a <table row value expression> is the declared type of the immediately contained <row value special case> or <row value constructor>.

4) The declared type of a <contextually typed row value expression> is the declared type of the immediately contained <row value special case> or <contextually typed row value constructor>. The declared type of a <row value predicand> is the declared type of the immediately contained <row value special case> or <row value constructor predicand>.

Access Rules

None.

General Rules

1) A <row value special case> specifies the row value denoted by the <nonparenthesized value expression primary>.

2) A <row value expression> specifies the row value denoted by the <row value special case> or <explicit row value constructor>.

3) A <table row value expression> specifies the row value denoted by the <row value special case> or <row value constructor>.

4) A <contextually typed row value expression> specifies the row value denoted by the <row value special case> or <contextually typed row value constructor>.

5) A <row value predicand> specifies the row value denoted by the <row value special case> or <row value constructor predicand>.

Conformance Rules

1) Without Feature T051, “Row types”, conforming SQL language shall not contain a <row value special case>.

7.3 <table value constructor>

Function

Specify a set of <row value expression>s to be constructed into a table.

Format

```
<table value constructor> ::=
  VALUES <row value expression list>

<row value expression list> ::=
  <table row value expression> [ { <comma> <table row value expression> }... ]

<contextually typed table value constructor> ::=
  VALUES <contextually typed row value expression list>

<contextually typed row value expression list> ::=
  <contextually typed row value expression>
      [ { <comma> <contextually typed row value expression> }... ]
```

Syntax Rules

1) All <table row value expression>s immediately contained in a <row value expression list> shall be of the same degree.

2) All <contextually typed row value expression>s immediately contained in a <contextually typed row value expression list> shall be of the same degree.

3) A <table value constructor> or a <contextually typed table value constructor> is *possibly non-deterministic* if it generally contains a possibly non-deterministic <value expression>.

4) Let *TVC* be some <table value constructor> consisting of *n* <table row value expression>s or some <contextually typed table value constructor> consisting of *n* <contextually typed row value expression>s. Let RVE_i, 1 (one) ≤ *i* ≤ *n*, denote the *i*-th <table row value expression> or the *i*-th <contextually typed row value expression>. The row type of *TVC* is determined by applying Subclause 9.3, "Result of data type combinations", to the row types RVE_i, 1 (one) ≤ *i* ≤ *n*. The column names are implementation-dependent.

Access Rules

None.

General Rules

1) If the result of any <table row value expression> or <contextually typed row value expression> is the null value, then an exception condition is raised: *data exception — null row not permitted in table*.

2) The result *T* of a <table value constructor> or <contextually typed table value constructor> *TVC* is a table whose cardinality is the number of <table row value expression>s or the number of <contextually typed row value expression>s in *TVC*. If *R* is the result of *n* such expressions, then *R* occurs *n* times in *T*.

Conformance Rules

1) Without Feature F641, “Row and table constructors”, in conforming SQL language, the <contextually typed row value expression list> of a <contextually typed table value constructor> shall contain exactly one <contextually typed row value constructor> *RVE*. *RVE* shall be of the form “(<contextually typed row value constructor element list>)”.

2) Without Feature F641, “Row and table constructors”, conforming SQL language shall not contain a <table value constructor>.

7.4 <table expression>

Function

Specify a table or a grouped table.

Format

```
<table expression> ::=
  <from clause>
      [ <where clause> ]
      [ <group by clause> ]
      [ <having clause> ]
      [ <window clause> ]
```

Syntax Rules

1) The result of a <table expression> is a derived table whose row type *RT* is the row type of the result of the application of last of the immediately contained <from clause>, <where clause>, <group by clause>, or <having clause> specified in the <table expression>, together with the window structure descriptors defined by the <window clause>, if specified.

2) Let *C* be some column. Let *TE* be the <table expression>. *C* is an underlying column of *TE* if and only if *C* is an underlying column of some column reference contained in *TE*.

Access Rules

None.

General Rules

1) If all optional clauses are omitted, then the result of the <table expression> is the same as the result of the <from clause>. Otherwise, each specified clause is applied to the result of the previously specified clause and the result of the <table expression> is the result of the application of the last specified clause.

Conformance Rules

None.

7.5 <from clause>

Function

Specify a table derived from one or more tables.

Format

```
<from clause> ::=
  FROM <table reference list>

<table reference list> ::=
  <table reference> [ { <comma> <table reference> }... ]
```

Syntax Rules

1) Let *TRL* be the ordering of <table reference list>. No element TR_i in *TRL* shall contain an outer reference to an element TR_j, where $i \leq j$.

2) Case:

   a) If the <table reference list> immediately contains a single <table reference>, then the descriptor of the result of the <table reference list> is the same as the descriptor of the table identified by that <table reference>. The row type *RT* of the result of the <table reference list> is the row type of the table identified by the <table reference>.

   b) If the <table reference list> immediately contains more than one <table reference>, then the descriptors of the columns of the result of the <table reference list> are the descriptors of the columns of the tables identified by the <table reference>s, in the order in which the <table reference>s appear in the <table reference list> and in the order in which the columns are defined within each table. The row type *RT* of the result of the <table reference list> is determined by the sequence *SCD* of column descriptors of the result as follows:

      i) Let *n* be the number of column descriptors in *SCD*. *RT* has *n* fields.

      ii) For *i* ranging from 1 (one) to *n*, the field name of the *i*-th field descriptor in *RT* is the column name included in the *i*-th column descriptor in *SCD*.

      iii) For *i* ranging from 1 (one) to *n*, the data type descriptor of the *i*-th field descriptor in *RT* is

      Case:

         1) If the *i*-th descriptor in *SCD* includes a domain name *DN*, then the data type descriptor included in the descriptor of the domain identified by *DN*.

         2) Otherwise, the data type descriptor included in the *i*-th column descriptor in *SCD*.

3) The descriptor of the result of the <from clause> is the same as the descriptor of the result of the <table reference list>.

Access Rules

None.

General Rules

1) Let *TRLR* be the result of *TRL*.

   Case:

   a) If *TRL* simply contains a single <table reference> *TR*, then *TRLR* is the result of *TR*.

   b) If *TRL* simply contains *n* <table reference>s, where $n > 1$, then let *TRLP* be the <table reference list> formed by taking the first $n-1$ elements of *TRL* in order, let *TRLL* be the last element of *TRL*, and let *TRLPR* be the result of *TRLP*. For every row R_i, 1 (one) $\leq i \leq n$, in *TRLPR*, let $TRLLR_i$ be the corresponding evaluation of *TRLL* under all outer references contained in *TRLL*. Let $SUBR_i$ be the table containing every row formed by concatenating R_i with some row of $TRLLR_i$. Every row *RR* in $SUBR_i$ is a row in *TRLR*, and the number of occurrences of *RR* in *TRLR* is the sum of the numbers of occurrences of *RR* in every occurrence of $SUBR_i$.

   The result of the <table reference list> is *TRLR* with the columns reordered according to the ordering of the descriptors of the columns of the <table reference list>.

2) The result of the <from clause> is *TRLR*.

Conformance Rules

None.

7.6 <table reference>

This Subclause is modified by Subclause 7.1, "<table reference>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 7.1, "<table reference>", in ISO/IEC 9075-14.

Function

Reference a table.

Format

```
<table reference> ::=
    <table factor>
  | <joined table>

<table factor> ::=
  <table primary> [ <sample clause> ]

<sample clause> ::=
  TABLESAMPLE <sample method> <left paren> <sample percentage> <right paren>
      [ <repeatable clause> ]

<sample method> ::=
    BERNOULLI
  | SYSTEM

<repeatable clause> ::=
  REPEATABLE <left paren> <repeat argument> <right paren>

<sample percentage> ::=
  <numeric value expression>

<repeat argument> ::=
  <numeric value expression>

<table primary> ::=
    <table or query name> [ [ AS ] <correlation name>
        [ <left paren> <derived column list> <right paren> ] ]
  | <derived table> [ AS ] <correlation name>
      [ <left paren> <derived column list> <right paren> ]
  | <lateral derived table> [ AS ] <correlation name>
      [ <left paren> <derived column list> <right paren> ]
  | <collection derived table> [ AS ] <correlation name>
      [ <left paren> <derived column list> <right paren> ]
  | <table function derived table> [ AS ] <correlation name>
      [ <left paren> <derived column list> <right paren> ]
  | <only spec> [ [ AS ] <correlation name>
      [ <left paren> <derived column list> <right paren> ] ]
  | <parenthesized joined table>

<only spec> ::=
  ONLY <left paren> <table or query name> <right paren>

<lateral derived table> ::=
  LATERAL <table subquery>
```

```
<collection derived table> ::=
  UNNEST <left paren> <collection value expression> <right paren>
      [ WITH ORDINALITY ]

<table function derived table> ::=
  TABLE <left paren> <collection value expression> <right paren>

<derived table> ::=
  <table subquery>

<table or query name> ::=
    <table name>
  | <transition table name>
  | <query name>

<derived column list> ::=
  <column name list>

<column name list> ::=
  <column name> [ { <comma> <column name> }... ]

<parenthesized joined table> ::=
    <left paren> <parenthesized joined table> <right paren>
  | <left paren> <joined table> <right paren>
```

Syntax Rules

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

2) If a <table primary> *TP* simply contains a <table function derived table> *TFDT*, then:

   a) The <collection value expression> immediately contained in *TFDT* shall be a <routine invocation>.

   b) Let *CN* be the <correlation name> simply contained in *TP*.

   c) Let *CVE* be the <collection value expression> simply contained in *TP*.

   d) Case:

      i) If *TP* specifies a <derived column list> *DCL*, then let *TFDCL* be

         (*DCL*)

      ii) Otherwise, let *TFDCL* be a zero-length string.

   e) *TP* is equivalent to the <table primary>

      UNNEST (*CVE*) AS *CN TFDCL*

3) If a <table primary> *TP* simply contains a <collection derived table> *CDT*, then let *CVE* be the <collection value expression> simply contained in *CDT*, let *CN* be the <correlation name> simply contained in *TP*, and let *TEMP* be an <identifier> that is not equivalent to *CN* nor to any other <identifier> contained in *TP*. Let *ET* be the element type of the declared type of *CVE*.

   a) Case:

i) If the declared type of *CVE* is a multiset, then WITH ORDINALITY shall not be specified. Let *IMDC* be the implementation-defined maximum cardinality of an array whose declared element type is *ET*. Let *C* be

```
( CAST (CVE AS ET ARRAY[IMDC]) )
```

ii) Otherwise, let *C* be *CVE*.

b) Let *N1* and *N2* be two <column name>s that are not equivalent to one another nor to *CN*, *TEMP*, or any other <identifier> contained in *TP*.

c) Let *RECQP* be:

```
WITH RECURSIVE TEMP(N1, N2) AS
  (   SELECT C[1] AS N1, 1 AS N2
      FROM (VALUES(1)) AS CN
      WHERE 0 < CARDINALITY(C)
    UNION
      SELECT C[N2+1] AS N1, N2+1 AS N2
      FROM TEMP
      WHERE N2 < CARDINALITY(C)
  )
```

d) Case:

i) If *TP* specifies a <derived column list> *DCL*, then:

1) Case:

A) If *CDT* specifies WITH ORDINALITY, then

Case:

I) If *ET* is a row type, then let *DET* be the degree of *ET*. *DCL* shall contain *DET*+1 (one) <column name>s.

II) Otherwise, *DCL* shall contain 2 <column name>s.

B) Otherwise,

Case:

I) If *ET* is a row type, then let *DET* be the degree of *ET*. *DCL* shall contain *DET* <column name>s.

II) Otherwise, *DCL* shall contain 1 (one) <column name>.

2) Let *PDCLP* be

```
( DCL )
```

ii) Otherwise,

Case:

1) If *ET* is a row type, then:

A) Let *DET* be the degree of *ET*.

B) Let FN_i, 1 (one) ≤ i ≤ *DET*, be the name of the i-th field in *ET*.

C) Case:

I) If *CDT* specifies WITH ORDINALITY, then let *PDCLP* be:

```
( FN1, FN2, ..., FNDET, N2 )
```

II) Otherwise, let *PDCLP* be:

```
( FN1, FN2, ..., FNDET )
```

2) Otherwise, let *PDCLP* be a zero-length string.

e) Case:

i) If *CDT* specifies WITH ORDINALITY, then

Case:

1) If *ET* is a row type, then let *ELDT* be:

```
LATERAL ( RECQP SELECT N1.*, N2
                FROM TEMP ) AS CN PDCLP
```

2) Otherwise, let *ELDT* be:

```
LATERAL ( RECQP SELECT *
                FROM TEMP ) AS CN PDCLP
```

ii) Otherwise,

Case:

1) If *ET* is a row type, then let *ELDT* be:

```
LATERAL ( RECQP SELECT N1.*
                FROM TEMP ) AS CN PDCLP
```

2) Otherwise, let *ELDT* be:

```
LATERAL ( RECQP SELECT N1
                FROM TEMP ) AS CN PDCLP
```

f) *TP* is equivalent to the <table primary> *ELDT*.

4) Let *TR* be a <table reference>.

Case:

a) If *TR* simply contains a <joined table>, then let *JT* be that <joined table>.

b) Otherwise, let *TF* be the <table factor> that is immediately contained in *TR*, and let *TP* be the <table primary> that is immediately contained in *TF*.

5) If *TF* simply contains a <correlation name>, then let *RV* be that <correlation name>; otherwise, let *RV* be the <table or query name> simply contained in *TF*. *RV* is a range variable. *RV* is *exposed* by *TF* and by *TR*.

NOTE 137 — "range variable" is defined in Subclause 4.14.6, "Operations involving tables".

NOTE 138 — Thus a range variable is a BNF nonterminal (either a <correlation name> or a <table or query name>). It is possible for an <SQL procedure statement> to contain several range variables whose <identifier>s are equivalent, which must nevertheless be regarded as distinct range variables.

6) Case:

   a) If *TR* is contained in a <from clause> *FC* with no intervening <query expression>, then the *scope clause SC* of *TR* is the <select statement: single row> or innermost <query specification> that contains *FC*. The scope of a range variable of *TR* is the <select list>, <where clause>, <group by clause>, <having clause>, and <window clause> of *SC*, together with every <lateral derived table> that is simply contained in *FC* and is preceded by *TR*, and every <collection derived table> that is simply contained in *FC* and is preceded by *TR*, and the <join condition> of all <joined table>s contained in *SC* that contain *TR*. If *SC* is the <query specification> that is the <query expression body> of a simple table query *STQ*, then the scope of a range variable of *TR* also includes the <order by clause> of *STQ*.

      NOTE 139 — "simple table query" is defined in Subclause 14.3, "<cursor specification>".

   b) If *TF* is simply contained in a <merge statement> *MS*, then the *scope clause SC* of *TR* is *MS*. The scope of the range variable of *TF* and of *TR* is the <search condition>, <set clause list>, and <merge insert value list> of *SC*.

      NOTE 140 — Subclause 14.11, "<merge statement>", does not allow *TR* to directly contain a <joined table>.

   c) Otherwise, the *scope clause SC* of *TR* is the outermost <joined table> that contains *TR* with no intervening <query expression>. The scope of any range variable of *TR* is the <join condition> of *SC* and of all <joined table>s contained in *SC* that contain *TR*.

7) Let *RV* be a range variable that is exposed by *TR*. Let *RV1* be a range variable that is exposed by a <table reference> *TR1* that has the same scope clause as *TR*.

   Case:

   a) If *RV* is a <table name>, then

      Case:

      i) If *RV1* is a <table name>, then *RV1* shall not be equivalent to *RV*.

      ii) Otherwise, *RV1* shall not be equivalent to the <qualified identifier> of *RV*.

   b) Otherwise,

      Case:

      i) If *RV1* is a <table name>, then the <qualified identifier> of *RV1* shall not be equivalent to *RV*.

      ii) Otherwise, *RV1* shall not be equivalent to *RV*.

8) A <table or query name> simply contained in *TF* has a scope clause and scope defined by *TF* if and only if the <table or query name> is exposed by *TF* and by *TR*.

9) If *TP* simply contains <table or query name> *TOQN*, then

   Case:

   a) If *TOQN* is an <identifier> that is equivalent to a <query name> *QN*, then let *WLE* be the <with list element> simply contained in the <query expression> that simply contains *TP* such that the <query

name> *QN1* simply contained in *WLE* is equivalent to *QN* and *QN1* is the innermost query name in scope. Let the *table specified by the <query name>* be the result of *WLE*.

NOTE 141 — "query name in scope" is defined in Subclause 7.13, "<query expression>".

b) If *TOQN* is an <identifier> that is equivalent to a <transition table name> that is in scope, then let the *table specified by the <transition table name>* be the table identified by *TOQN*.

NOTE 142 — The scope of a <transition table name> is defined in Subclause 11.39, "<trigger definition>".

c) Otherwise, let the *table specified by the <table name>* be the table identified by the <table name> simply contained in *TP*.

NOTE 143 — The preceding cases disambiguate whether *TOQN* is interpreted as a <query name>, <transition table name>, or <table name>.

10) If *TP* simply contains <only spec> *OS* and the table identified by the <table or query name> *TN* is not a typed table, then *OS* is equivalent to *TN*.

11) No <column name> shall be specified more than once in a <derived column list>.

12) If a <derived column list> is specified in *TP*, then the number of <column name>s in the <derived column list> shall be the same as the degree of the table specified by the <derived table>, the <lateral derived table>, or the <table or query name> simply contained in *TP*, and the name of the *i*-th column of that <derived table> or <lateral derived table> or the effective name of the *i*-th column of that <table or query name> is the *i*-th <column name> in that <derived column list>.

13) The row type of a <lateral derived table> is the row type of the simply contained <query expression>.

14) Case:

a) If no <derived column list> is specified in *TR*, then the row type *RT* of *TR* is the row type of its simply contained <table or query name>, <derived table>, <lateral derived table>, or <joined table>.

b) Otherwise, the row type *RT* of *TR* is described by a sequence of (<field name>, <data type>) pairs, where the <field name> in the *i*-th pair is the *i*-th <column name> in the <derived column list> and the <data type> in the *i*-th pair is the declared type of the *i*-th column of the <derived table>, <joined table>, <lateral derived table>, or of the table identified by the <table or query name> simply contained in *TR*.

15) A <derived table> or <lateral derived table> is an *updatable derived table* if and only if the <query expression> simply contained in the <derived table> or <lateral derived table> is updatable.

16) A <derived table> or <lateral derived table> is a *simply updatable derived table* if and only if the <query expression> simply contained in the <derived table> or <lateral derived table> simply is updatable.

17) A <derived table> or <lateral derived table> is an *insertable-into derived table* if and only if the <query expression> simply contained in the <derived table> or <lateral derived table> is insertable-into.

18) A <collection derived table> is not updatable and is not simply updatable.

19) If a <table reference> *TR* immediately contains a <table factor> *TF*, then

Case:

a) If *TF* simply contains a <table name> that identifies a base table, then every column of the table identified by *TF* is called an *updatable column* of *TR*.

b) If *TF* simply contains a <table name> that identifies a view, then every updatable column of the view identified by *TF* is called an *updatable column* of *TR*.

c) If *TF* simply contains a <derived table> or <lateral derived table>, then every updatable column of the table identified by the <query expression> simply contained in <derived table> or <lateral derived table> is called an *updatable column* of *TR*.

20) If a <table reference> *TR* immediately contains a <table factor> and the <table or query name> simply contained in *TR* immediately contains a <table name> *TN*, then let *T* be the table identified by *TN*. The schema identified by the explicit or implicit qualifier of *TN* shall include the descriptor of *T*.

21) A <table name> is *possibly non-deterministic* if the table identified by the <table name> is a viewed table, and the original <query expression> in the view descriptor identified by the <table name> is possibly non-deterministic.

22) A <query name> is *possibly non-deterministic* if the <query expression> identified by the <query name> is possibly non-deterministic.

23) A <derived table> or <lateral derived table> is *possibly non-deterministic* if the simply contained <query expression> is possibly non-deterministic.

24) A <table primary> is *possibly non-deterministic* if the simply contained <table name>, <query name>, <derived table>, <lateral derived table>, or <joined table> is possibly non-deterministic.

25) A <table reference> is *possibly non-deterministic* if the simply contained <table primary> or <joined table> is possibly non-deterministic or if <sample clause> is specified.

Access Rules

1) If a <table primary> *TP* simply contains a <table or query name> that simply contains a <table name> *TN*, then:

   a) Let *T* be the table identified by *TN*.

   b) Case:

      i) If *TN* is contained in a <search condition> immediately contained in an <assertion definition>or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on at least one column of *T*.

      ii) Otherwise:

         1) Case:

            A) If *TP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on at least one column of *T*.

            B) Otherwise, the current privileges shall include SELECT on at least one column of *T*.

         2) If *TP* simply contains <only spec> and *TN* identifies a typed table, then

            Case:

            A) If *TP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

B) Otherwise, the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

General Rules

1) If a <table primary> *TP* simply contains a <table or query name> *TOQN*, then

   Case:

   a) If *TOQN* simply contains a <query name> *QN*, then the result of *TP* is the table specified by *QN*.

   b) If *TOQN* simply contains a <transition table name> *TTN*, then the result of *TP* is the table specified by *TTN*.

      NOTE 144 — The table identified by a <transition table name> is a transition table as defined by the General Rules of Subclause 15.7, "Effect of deleting rows from base tables", Subclause 15.10, "Effect of inserting tables into base tables", or Subclause 15.13, "Effect of replacing rows in base tables", as appropriate.

   c) Otherwise, let *T* be the table specified by the <table name> simply contained in *TP*.

      Case:

      i) If ONLY is specified, then the result of *TP* is a table that consists of every row in *T*, except those rows that have a subrow in a proper subtable of *T*.

      ii) Otherwise, the result of *TP* is a table that consists of every row of *T*.

2) If a <derived table> or <lateral derived table> *LDT* simply containing <query expression> *QE* is specified, then the result of *LDT* is the result of *QE*.

3) If *TP* immediately contains a <joined table>, then the result of *TP* is the result of that <joined table>.

4) Let *TP* be the <table primary> immediately contained in a <table factor> *TF*. Let *RT* be the result of *TP*.

   Case:

   a) If <sample clause> is specified, then:

      i) Let *N* be the number of rows in *RT* and let *S* be the value of <sample percentage>.

      ii) If *S* is the null value or if *S* < 0 (zero) or if *S* > 100, then an exception condition is raised: *data exception — invalid sample size*.

      iii) If <repeatable clause> is specified, then let *RPT* be the value of <repeat argument>. If *RPT* is the null value, then an exception condition is raised: *data exception — invalid repeat argument in a sample clause*.

      iv) Case:

         1) If <sample method> specifies BERNOULLI, then the result of *TF* is a table containing approximately (*N***S*/100) rows of *RT*. The probability of a row of *RT* being included in result of *TF* is *S*/100. Further, whether a given row of *RT* is included in result of *TF* is independent of whether other rows of *RT* are included in result of *TF*.

         2) Otherwise, result of *TF* is a table containing approximately (*N***S*/100) rows of *RT*. The probability of a row of *RT* being included in result of *TF* is *S*/100.

v) If *TF* contains outer references, then a table with identical rows is generated every time *TF* is evaluated with a given set of values for outer references.

NOTE 145 — "outer reference" is defined in Subclause 6.7, "<column reference>".

b) Otherwise, result of *TF* is *RT*.

5) The result of a <table reference> *TR* is the result of immediately contained <table factor> or <joined table>.

6) Let *RV* be the range variable that is exposed by a <table factor> *TF*. The table associated with *RV* is the result of *TF*.

NOTE 146 — "table associated with *RV*" is defined in Subclause 4.14.6, "Operations involving tables".

Conformance Rules

1) Without Feature S091, "Basic array support", or Feature S271, "Basic multiset support", conforming SQL language shall not contain a <collection derived table>.

2) Without Feature T491, "LATERAL derived table", conforming SQL language shall not contain a <lateral derived table>.

3) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", conforming SQL language shall not contain a <query name>.

4) Without Feature S111, "ONLY in query expressions", conforming SQL language shall not contain a <table reference> that contains an <only spec>.

5) Without Feature F591, "Derived tables", conforming SQL language shall not contain a <derived table>.

6) Without Feature T326, "Table functions", conforming SQL language shall not contain a <table function derived table>.

7) Without Feature T613, "Sampling", conforming SQL language shall not contain a <sample clause>.

8) Without Feature T211, "Basic trigger capability", conforming SQL language shall not contain a <transition table name>.

7.7 <joined table>

Function

Specify a table derived from a Cartesian product, inner join, or outer join.

Format

```
<joined table> ::=
    <cross join>
  | <qualified join>
  | <natural join>

<cross join> ::=
  <table reference> CROSS JOIN <table factor>

<qualified join> ::=
  { <table reference> | <partitioned join table> }
      [ <join type> ] JOIN
      { <table reference> | <partitioned join table> }
      <join specification>

<partitioned join table> ::=
  <table factor> PARTITION BY
      <partitioned join column reference list>

<partitioned join column reference list> ::=
  <left paren> <partitioned join column reference>
      [ { <comma> <partitioned join column reference> }... ]
      <right paren>

<partitioned join column reference> ::=
  <column reference>

<natural join> ::=
  { <table reference> | <partitioned join table> }
      NATURAL [ <join type> ] JOIN
      { <table factor> | <partitioned join table> }

<join specification> ::=
    <join condition>
  | <named columns join>

<join condition> ::=
  ON <search condition>

<named columns join> ::=
  USING <left paren> <join column list> <right paren>

<join type> ::=
    INNER
  | <outer join type> [ OUTER ]

<outer join type> ::=
    LEFT
  | RIGHT
```

```
  | FULL

<join column list> ::=
  <column name list>
```

Syntax Rules

1) Let *TRA* be the <table reference> or <table factor> that is the first operand of the <joined table>, and let *TRB* be the <table reference> or <table factor> that is the second operand of the <joined table>. Let *RTA* and *RTB* be the row types of *TRA* and *TRB*, respectively. Let *RVA* and *RVB* be the range variables of *TRA* and *TRB*, respectively. Let *CP* be:

   ```
   SELECT *
   FROM TRA, TRB
   ```

2) If *TRB* contains a <lateral derived table> containing an outer reference that references *TRA*, then <join type> shall not contain RIGHT or FULL.

3) If a <qualified join> or <natural join> is specified and a <join type> is not specified, then INNER is implicit.

4) If a <qualified join> containing a <join condition> is specified and a <value expression> directly contained in the <search condition> is a <set function specification>, then the <joined table> shall be contained in a <having clause> or <select list>, the <set function specification> shall contain an aggregated argument *AA* that contains an outer reference, and every column reference contained in *AA* shall be an outer reference.

   NOTE 147 — "outer reference" is defined in Subclause 6.7, "<column reference>".

5) The <search condition> shall not contain a <window function> without an intervening <query expression>.

6) If neither NATURAL is specified nor a <join specification> immediately containing a <named columns join> is specified, then the descriptors of the columns of the result of the <joined table> are the same as the descriptors of the columns of *CP*, with the possible exception of the nullability characteristics of the columns.

7) If a <partitioned join table> *PJT* is specified, then:

   a) The <qualified join> or <natural join> shall specify an <outer join type>.

   b) Each <partitioned join column reference> shall uniquely reference a column of the table referenced by the <table primary> simply contained in *PJT*. Such a column is called a *join partitioning column*.

   c) If the first operand of the <qualified join> or <natural join> is a <partitioned join table>, then the <outer join type> shall be RIGHT or FULL.

   d) If the second operand of the <qualified join> or <natural join> is a <partitioned join table>, then the <outer join type> shall be LEFT or FULL, and *TRB* shall not contain a <lateral derived table> containing an outer reference that references *TRA*.

8) If NATURAL is specified or if a <join specification> immediately containing a <named columns join> is specified, then:

   a) If NATURAL is specified, then let *common column name* be a <field name> that is equivalent to the <field name> of exactly one field of *RTA* and the <field name> of exactly one field of *RTB*. *RTA* shall not have any duplicate common column names and *RTB* shall not have any duplicate common column names. Let *corresponding join columns* refer to all fields of *RTA* and *RTB* that have common column names, if any.

b) If a <named columns join> is specified, then every <column name> in the <join column list> shall be equivalent to the <field name> of exactly one field of *RTA* and the <field name> of exactly one field of *RTB*. Let *common column name* be the name of such a column. Let *corresponding join columns* refer to the columns identified in the <join column list>.

c) A corresponding join column shall not be a join partitioning column.

d) Let *CA* and *CB* be a pair of corresponding join columns of *RTA* and *RTB*, respectively. *CA* and *CB* shall be comparable. *CA* and *CB* are operands of an equality operation, and the Syntax Rules of Subclause 9.9, "Equality operations", apply.

e) If there is at least one corresponding join column, then let *SLCC* be a <select list> of <derived column>s of the form

```
COALESCE ( RVA.C, RVB.C ) AS C
```

for every column *C* that is a corresponding join column, taken in order of their ordinal positions in *RTA*.

f) If *RTA* contains at least one field that is not a corresponding join column, then let *SLTA* be a <select list> of <derived column>s of the form

```
RVA.C
```

for every field *C* of *RTA* that is not a corresponding join column, taken in order of their ordinal positions in *RTA*.

g) If *RTB* contains at least one field that is not a corresponding join column, then let *SLTB* be a <select list> of <derived column>s of the form

```
RVB.C
```

for every field *C* of *RTB* that is not a corresponding join column, taken in order of their ordinal positions in *RTB*.

h) Let the <select list> *SL* be defined as

Case:

i) If all of the fields of *RTA* and *RTB* are corresponding join columns, then let *SL* be "*SLCC*".

ii) If *RTA* contains no corresponding join columns and *RTB* contains no corresponding join columns, then let *SL* be "*SLTA*, *SLTB*".

iii) If *RTA* contains no fields other than corresponding join columns, then let *SL* be "*SLCC*, *SLTB*".

iv) If *RTB* contains no fields other than corresponding join columns, then let *SL* be "*SLCC*, *SLTA*".

v) Otherwise, let *SL* be "*SLCC*, *SLTA*, *SLTB*".

The descriptors of the columns of the result of the <joined table>, with the possible exception of the nullability characteristics of the columns, are the same as the descriptors of the columns of the result of

```
SELECT SL FROM TRA, TRB
```

9) A <joined table> is *possibly non-deterministic* if at least one of the following conditions is true:

a) Either *TRA* or *TRB* is possibly non-deterministic.

b) A <join condition> that generally contains a possibly non-deterministic <value expression>, possibly non-deterministic <query specification>, or possibly non-deterministic <query expression> is specified.

c) NATURAL is specified, or a <join specification> immediately containing a <named columns join> is specified, and there is a common column name *CCN* such that the declared types of the two corresponding join columns identified by *CCN* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

d) The declared type of a join partitioning column is character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE.

10) The declared type of the rows of the <joined table> is the row type *RT* defined by the sequence of (<field name>, <data type>) pairs indicated by the sequence of column descriptors of the <joined table> taken in order.

11) For every column *CR* of the result of the <joined table> that corresponds to a field *CA* of *RTA* that is not a corresponding join column or a join partitioning column, *CR* is *possibly nullable* if any of the following conditions are true:

a) RIGHT or FULL is specified.

b) INNER, LEFT, or CROSS JOIN is specified or implicit and *CA* is possibly nullable.

12) For every column *CR* of the result of the <joined table> that corresponds to a field *CB* of *RTB* that is not a corresponding join column or a join partitioning column, *CR* is *possibly nullable* if any of the following conditions are true:

a) LEFT or FULL is specified.

b) INNER, RIGHT, or CROSS JOIN is specified or implicit and *CB* is possibly nullable.

13) For every column *CR* of the result of the <joined table> that corresponds to a corresponding join column *CA* of *RTA* and a corresponding join column *CB* of *RTB*, *CR* is *possibly nullable* if any of the following conditions are true:

a) LEFT or FULL is specified and *CA* is possibly nullable, or

b) RIGHT or FULL is specified and *CB* is possibly nullable.

14) A column *CR* of the result of the <joined table> that corresponds to a join partitioning column *JPC* is *possibly nullable* if *JPC* is possibly nullable.

Access Rules

None.

General Rules

1) Case:

a) If a <cross join> is specified, then let *T* be *CP*.

b) If a <join condition> is specified, then let *SC* be the <search condition> and let *T* be

```
CP
WHERE SC
```

c) If NATURAL is specified or <named columns join> is specified, then

Case:

i) If there are corresponding join columns, then let N be the number of corresponding join columns and let $CJCN_i$, 1 (one) $\leq i \leq N$, be the field name of the i-th corresponding join column, and let T be

```
CP
WHERE RVA.CJCN1 = RVB.CJCN1
  AND ...
  AND RVA.CJCNN = RVB.CJCNN
```

ii) Otherwise, let T be CP.

2) Let TR be the result of evaluating T, let DA and DB be the degrees of TRA and TRB, respectively, and let TVA be the result of evaluating TRA.

3) If LEFT is specified and the second operand does not specify <partitioned join table>, then:

a) Let PA be the collection of rows of TVA for which there exists some row R in TR and some row RA in TVA such that the values of the first DA fields of R are identical to the values of the corresponding fields of RA.

b) Let UA be those rows of TVA that are not in PA.

c) Let XA be UA extended on the right with DB columns containing the null value.

d) Let XNA be an effective distinct name for XA. Let TN be an effective name for T.

4) If RIGHT or FULL is specified or if LEFT is specified and the second operand specifies <partitioned join table>, then:

a) Let TVB be the result of evaluating TRB.

NOTE 148 — It follows from the Syntax Rules that TRB does not contain a <lateral derived table> containing an outer reference that references TRA. This ensures that it is possible to evaluate TRB in isolation.

b) Case:

i) If the first operand specifies <partitioned join table>, then

Case:

1) If TVA is empty, then let NA be 0 (zero).

2) Otherwise, TVA is partitioned into the minimum numbers of partitions such that for each join partitioning column JPC of each partition, no two values of JPC are distinct. If the declared type of a join partitioning column is a user-defined type and the comparison of that column results in <u>*Unknown*</u> for two rows of TVA, then the assignment of those rows to partitions is implementation-dependent. Let NA be the number of partitions. Let $GA_1, \ldots, GA_{NA}$ be an enumeration of the partitions.

ii) Otherwise, let *NA* be 1 (one), and let GA_1 be *TVA*.

c) Case:

i) If the second operand specifies <partitioned join table>, then Case:

1) If *TVB* is empty, then let *NB* be 0 (zero).

2) Otherwise, *TVB* is partitioned into the minimum numbers of partitions such that for each join partitioning column *JPC* of each partition, no two values of the join partitioning column are distinct. If the declared type of a join partitioning column is a user-defined type and the comparison of that column results in <u>*Unknown*</u> for two rows of *TVB*, then the assignment of those rows to partitions is implementation-dependent. Let *NB* be the number of partitions. Let GB_1, ... , GB_{NB} be an enumeration of the partitions.

ii) Otherwise, let *NB* be 1 (one), and let GB_1 be *TVB*.

d) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $PA_{i,\,j}$ be the collection of rows *RA* of GA_i for which there exists a row *RB* in GB_j such that the concatenation of *RA* and *RB* is in *TR*.

e) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $PB_{i,\,j}$ be the collection of rows *RB* of GB_i for which there exists a row *RA* in GA_j such that the concatenation of *RA* and *RB* is in *TR*.

f) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $UA_{i,\,j}$ be the collection of rows of GA_i that are not in $PA_{i,\,j}$.

g) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $UB_{i,\,j}$ be the collection of rows of GB_i that are not in $PB_{i,\,j}$.

h) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $XA_{i,\,j}$ be $UA_{i,\,j}$ extended on the right with *DB* columns, with declared types and values determined as follows. For each *k* between 1 (one) and *DB*, the declared type of the (*DA* + *k*)-th column is the declared type of the *k*-th column of *TVB*, and the value is

Case:

i) If the *k*-th column of *TVB* is a join partitioning column, then the common value of the *k*-th column of GB_j.

ii) Otherwise, the null value.

i) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $XB_{i,\,j}$ be $UB_{i,\,j}$ extended on the left with *DA* columns, with declared types and values determined as follows. For each *k* between 1 (one) and *DA*, the declared type of the *k*-th column is the declared type of the *k*-th column of *TVA*, and the value is

Case:

i) If the *k*-th column of *TVA* is a join partitioning column, then the common value of the *k*-th column of GA_i.

ii) Otherwise, the null value.

j) Let *XA* be the collection of all rows in $XA_{i,j}$ for all *i* between 1 (one) and *NA* and all *j* between 1 (one) and *NB*.

NOTE 149 — If *NA* is 0 (zero), then *XA* is empty.

k) Let *XB* be the collection of all rows in $XB_{i,j}$, for all *i* between 1 (one) and *NA* and all *j* between 1 (one) and *NB*.

NOTE 150 — If *NB* is 0 (zero), then *XB* is empty.

l) Let *XNB* be an effective distinct name for *XB*.

5) Case:

a) If INER or <cross join> is specified, then let *S* be *TR*.

b) If LEFT is specified, then let *S* be the result of:

```
SELECT *
FROM TN
UNION ALL
SELECT *
FROM XNA
```

c) If RIGHT is specified, then let *S* be the result of:

```
SELECT *
FROM TN
UNION ALL
SELECT *
FROM XNB
```

d) If FULL is specified, then let *S* be the result of:

```
SELECT *
FROM TN
UNION ALL
SELECT *
FROM XNA
UNION ALL
SELECT *
FROM XNB
```

6) Let *SN* be an effective name of *S*.

Case:

a) If NATURAL is specified or a <named columns join> is specified, then:

i) Let CS_i be a name for the *i*-th column of *S*. Column CS_i of *S* corresponds to the *i*-th field of *RTA* if *i* is less than or equal to *DA*. Column CS_j of *S* corresponds to the (*j*–*DA*)-th field of *RTB* for *j* greater than *DA*.

ii) If there is at least one corresponding join column, then let *SLCC* be a <select list> of derived columns of the form

```
COALESCE (CS_i, CS_j)
```

for every pair of columns CS_i and CS_j, where CS_i and CS_j correspond to fields of *RTA* and *RTB* that are a pair of corresponding join columns.

iii) If *RTA* contains one or more fields that are not corresponding join columns, then let *SLTA* be a <select list> of the form:

```
CSi
```

for every column CS_i of *S* that corresponds to a field of *RTA* that is not a corresponding join column, taken in order of their ordinal position in *S*.

iv) If *RTB* contains one or more fields that are not corresponding join columns, then let *SLTB* be a <select list> of the form:

```
CSj
```

for every column CS_j of *S* that corresponds to a field of *RTB* that is not a corresponding join column, taken in order of their ordinal position in *S*.

v) Let the <select list> *SL* be defined as

Case:

1) If all the fields of *RTA* and *RTB* are corresponding join columns, then let *SL* be

```
SLCC
```

2) If *RTA* contains no corresponding join columns and *RTB* contains no corresponding join columns, then let *SL* be

```
SLTA, SLTB
```

3) If *RTA* contains no fields other than corresponding join columns, then let *SL* be

```
SLCC, SLTB
```

4) If *RTB* contains no fields other than corresponding join columns, then let *SL* be

```
SLCC, SLTA
```

5) Otherwise, let *SL* be

```
SLCC, SLTA, SLTB
```

vi) The result of the <joined table> is the result of:

```
SELECT SL
FROM SN
```

b) Otherwise, the result of the <joined table> is *S*.

Conformance Rules

1) Without Feature F401, “Extended joined table”, conforming SQL language shall not contain a <cross join>.

2) Without Feature F401, “Extended joined table”, conforming SQL language shall not contain a <natural join>.

3) Without Feature F401, “Extended joined table”, conforming SQL language shall not contain FULL.

4) Without Feature F402, “Named column joins for LOBs, arrays, and multisets”, conforming SQL language shall not contain a <joined table> that simply contains either <natural join> or <named columns join> in which, if *C* is a corresponding join column, the declared type of *C* is LOB-ordered, array-ordered, or multiset-ordered.

   NOTE 151 — If *C* is a corresponding join column, then the Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

5) Without Feature F403, “Partitioned join tables”, conforming SQL language shall not contain <partitioned join table>.

7.8 <where clause>

Function

Specify a table derived by the application of a <search condition> to the result of the preceding <from clause>.

Format

```
<where clause> ::=
  WHERE <search condition>
```

Syntax Rules

1) If a <value expression> directly contained in the <search condition> is a <set function specification>, then the <where clause> shall be contained in a <having clause> or <select list>, the <set function specification> shall contain a column reference, and every column reference contained in an aggregated argument of the <set function specification> shall be an outer reference.

   NOTE 152 — *outer reference* is defined in Subclause 6.7, "<column reference>".

2) The <search condition> shall not contain a <window function> without an intervening <query expression>.

Access Rules

None.

General Rules

1) Let *T* be the result of the preceding <from clause>.

2) The <search condition> is effectively evaluated for each row of *T*. The result of the <where clause> is a table of those rows of *T* for which the result of the <search condition> is *True*.

Conformance Rules

1) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <value expression> directly contained in a <where clause> that contains a <column reference> that references a <derived column> that generally contains a <set function specification> without an intervening <routine invocation>.

7.9 <group by clause>

Function

Specify a grouped table derived by the application of the <group by clause> to the result of the previously specified clause.

Format

```
<group by clause> ::=
  GROUP BY [ <set quantifier> ] <grouping element list>

<grouping element list> ::=
  <grouping element> [ { <comma> <grouping element> }... ]

<grouping element> ::=
    <ordinary grouping set>
  | <rollup list>
  | <cube list>
  | <grouping sets specification>
  | <empty grouping set>

<ordinary grouping set> ::=
    <grouping column reference>
  | <left paren> <grouping column reference list> <right paren>

<grouping column reference> ::=
  <column reference> [ <collate clause> ]

<grouping column reference list> ::=
  <grouping column reference> [ { <comma> <grouping column reference> }... ]

<rollup list> ::=
  ROLLUP <left paren> <ordinary grouping set list> <right paren>

<ordinary grouping set list> ::=
  <ordinary grouping set> [ { <comma> <ordinary grouping set> }... ]

<cube list> ::=
  CUBE <left paren> <ordinary grouping set list> <right paren>

<grouping sets specification> ::=
  GROUPING SETS <left paren> <grouping set list> <right paren>

<grouping set list> ::=
  <grouping set> [ { <comma> <grouping set> }... ]

<grouping set> ::=
    <ordinary grouping set>
  | <rollup list>
  | <cube list>
  | <grouping sets specification>
  | <empty grouping set>

<empty grouping set> ::=
  <left paren> <right paren>
```

Syntax Rules

1) Each <grouping column reference> shall unambiguously reference a column of the table resulting from the <from clause>. A column referenced in a <group by clause> is a *grouping column*.

   NOTE 153 — "Column reference" is defined in Subclause 6.7, "<column reference>".

2) Each <grouping column reference> is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

3) For every <grouping column reference> *GC*,

   Case:

   a) If <collate clause> is specified, then let *CS* be the collation identified by <collation name>. The declared type of the column reference shall be character string. The declared type of *GC* is that of its column reference, except that *CS* is the declared type collation and the collation derivation is *explicit*.

   b) Otherwise, the declared type of *GC* is the declared type of its column reference.

4) Let *QS* be the <query specification> that simply contains the <group by clause>, and let *SL*, *FC*, *WC*, *GBC*, and *HC* be the <select list>, the <from clause>, the <where clause> if any, the <group by clause>, and the <having clause> if any, respectively, that are simply contained in *QS*.

5) Let *QSSQ* be the explicit or implicit <set quantifier> immediately contained in *QS*.

6) Let *GBSQ* be the <set quantifier> immediately contained in <group by clause>, if any; otherwise, let *GBSQ* be ALL.

7) Let *SL1* be obtained from *SL* by replacing every <asterisk> and <asterisked identifier chain> using the syntactic transformations in the Syntax Rules of Subclause 7.12, "<query specification>".

8) A <group by clause> is *primitive* if it does not contain a <rollup list>, <cube list>, <grouping sets specification>, or <grouping column reference list>, and does not contain both a <grouping column reference> and an <empty grouping set>.

9) A <group by clause> is *simple* if it does not contain a <rollup list>, <cube list> or <grouping sets specification>.

10) If *GBC* is a simple <group by clause> that is not primitive, then *GBC* is transformed into a primitive <group by clause> as follows:

    a) Let *NSGB* be the number of <grouping column reference>s contained in *GBC*.

    b) Case:

       i) If *NSGB* is 0 (zero), then *GBC* is replaced by

          ```
          GROUP BY ()
          ```

       ii) Otherwise:

          1) Let $SGCR_1$, ... $SGCR_{NSGB}$ be an enumeration of the <grouping column reference>s contained in *GBC*.

          2) *GBC* is replaced by

```
GROUP BY SGCR1, ...
SGCRNSGB
```

NOTE 154 — That is, a simple <group by clause> that is not primitive may be transformed into a primitive <group by clause> by deleting all parentheses, and deleting extra <comma>s as necessary for correct syntax. If there are no grouping columns at all (for example, `GROUP BY (), ()`), this is transformed to the canonical form `GROUP BY ()`.

11) If *GBC* is a primitive <group by clause>, then let *SLNEW* and *HCNEW* be obtained from *SL1* and *HC*, respectively, by replacing every <grouping operation> by the exact numeric literal 0 (zero). *QS* is equivalent to:

```
SELECT QSSQ SLNEW FC
WC GBC HCNEW
```

12) If *OGSL* is an <ordinary grouping set list>, then the *concatenation* of *OGSL* is defined as follows:

a) Let *NGCR* be the number of <grouping column reference>s simply contained in *OGSL* and let GCR_j, 1 (one) $\leq j \leq NGCR$, be an enumeration of those <grouping column reference>s, in order from left to right.

b) The concatenation of *OGSL* is the <ordinary grouping set list>

```
GCR1, ..., GCRNGCR
```

NOTE 155 — Thus, the concatenation of *OGSL* may be formed by erasing all parentheses. For example, the concatenation of "(A, B), (C, D)" is "A, B, C, D".

13) If *RL* is a <rollup list>, then let OGS_i range over the *n* <ordinary grouping set>s contained in *RL*.

a) For each *i* between 1 (one) and *n*, let $COGS_i$ be the concatenation of the <ordinary grouping set list>

```
ORG1, ORG2,
..., ORGi
```

b) *RL* is equivalent to:

```
GROUPING SETS (
   ( COGSn ),
   ( COGSn-1 ),
   ( COGSn-2 ),
   ...
   ( COGS1 ),
   ( ) )
```

NOTE 156 — The result of the transform is to replace *RL* with a <grouping sets specification> that contains a <grouping set> for every initial sublist of the <ordinary grouping set list> of the <rollup list>, obtained by dropping <ordinary grouping set>s from the right, one by one, and concatenating each <ordinary grouping set list> so obtained. The <empty grouping set> is regarded as the shortest such initial sublist. For example, "ROLLUP ((A, B), (C, D))" is equivalent to "GROUPING SETS ((A, B, C, D), (A, B), ())".

14) If *CL* is a <cube list>, then let OGS_i range over the *n* <ordinary grouping set>s contained in *CL*. *CL* is transformed as follows:

a) Let $M = 2^n - 1$ (one).

b) For each i between 1 (one) and M:

i) Let BSL_i be the binary number consisting of n bits (binary digits) whose value is i.

ii) For each j between 1 (one) and n, let $B_{i,j}$ be the j-th bit, counting from left to right, in BSL_i.

iii) For each j between 1 (one) and n, let $GSLCR_{i,j}$ be

Case:

1) If $B_{i,j}$ is 0 (zero), then the zero-length string.

2) If $B_{i,j}$ is 1 (one) and $B_{i,k}$ is 0 (zero) for all $k < j$, then OGS_j.

3) Otherwise, <comma> followed by OGS_j.

iv) Let GSL_i be the concatenation of the <ordinary grouping set list>

$GSLCR_{i,1}$ $GSLCR_{i,2}$ ... $GSLCR_{i,n}$

c) *CL* is equivalent to

GROUPING SETS ((GSL_M), (GSL_{M-1}), ..., (GSL_1), ( ))

NOTE 157 — The result of the transform is to replace *CL* with a <grouping sets specification> that contains a <grouping set> for all possible subsets of the set of <ordinary grouping set>s in the <ordinary grouping set list> of the <cube list>, including <empty grouping set> as the empty subset with no <ordinary grouping set>s.

For example, `CUBE (A, B, C)` is equivalent to:

```
GROUPING SETS ( /* BSLi */
  (A, B, C),    /* 111 */
  (A, B   ),    /* 110 */
  (A,    C),    /* 101 */
  (A      ),    /* 100 */
  (   B, C),    /* 011 */
  (   B   ),    /* 010 */
  (      C),    /* 001 */
  (       )
)
```

As another example, `CUBE ((A, B), (C, D))` is equivalent to:

```
GROUPING SETS ( /* BSLi */
  (A, B, C, D), /* 11 */
  (A, B      ), /* 10 */
  (      C, D), /* 01 */
  (          )
)
```

15) If <grouping sets specification> *GSSA* simply contains another <grouping sets specification> *GSSB*, then *GSSA* is transformed as follows:

a) Let *NA* be the number of <grouping set>s simply contained in *GSSA*, and let *NB* be the number of <grouping set>s simply contained in *GSSB*.

b) Let GSA_i be an enumeration of the <grouping set>s simply contained in *GSSA*, for 1 (one) ≤ i ≤ *NA*.

c) Let GSB_i be an enumeration of the <grouping set>s simply contained in *GSSB*, 1 (one) ≤ i ≤ *NB*.

d) Let k be the value such that $GSSB = GSA_k$.

e) *GSSA* is equivalent to

```
GROUPING SETS (
    GSA1, GSA2, ... GSAk-1,
    GSB1, ... , GSBNB,
    GSAk+1, ..., GSANA )
```

NOTE 158 — Thus, the nested <grouping sets specification> is removed by simply "promoting" each of its <grouping set>s to be a <grouping set> of the encompassing <grouping sets specification>.

16) If *CGB* is a <group by clause> that is not simple, then *CGB* is transformed as follows:

a) The preceding Syntax Rules are applied repeatedly to eliminate any <grouping sets specification> that is nested in another <grouping sets specification>, as well as any <rollup list> and any <cube list>.

NOTE 159 — As a result, *CGB* is a list of two or more <grouping set>s, each of which is an <ordinary grouping set>, an <empty grouping set>, or a <grouping sets specification> that contains only <ordinary grouping set>s and <empty grouping set>s. There are no remaining <rollup list>s, <cube list>s, or nested <grouping sets specification>s.

b) Any <grouping element> *GS* that is an <ordinary grouping set> or an <empty grouping set> is replaced by the <grouping sets specification>

```
GROUPING SETS ( GS )
```

NOTE 160 — As a result, *CGB* is a list of two or more <grouping sets specification>s.

c) Let *GSSX* and *GSSY* be the first two <grouping sets specification>s in *CGB*. *CGB* is transformed by replacing "*GSSX* <comma> *GSSY*" as follows:

i) Let *NX* be the number of <grouping set>s in *GSSX* and let *NY* be the number of <grouping set>s in *GSSY*.

ii) Let GSX_i, 1 (one) ≤ i ≤ *NX*, be the <grouping set>s contained in *GSSX*, and let GSY_i, 1 (one) ≤ i ≤ *NY*, be the <grouping set>s contained in *GSSY*.

iii) Let $MX(i)$ be the number of <grouping column reference>s in GSX_i, and let $MY(i)$ be the number of <grouping column reference>s in GSY_i.

NOTE 161 — If GSX_i is <empty grouping set>, then $MX(i)$ is 0 (zero); and similarly for GSY_i.

iv) Let $GCRX_{i,j}$, 1 (one) ≤ j ≤ $MX(i)$ be the <grouping column reference>s contained in GSX_i, and let $GCRY_{i,j}$, 1 (one) ≤ j ≤ $MY(i)$ be the <grouping column reference>s contained in GSY_i.

NOTE 162 — If GSX_i is <empty grouping set>, then there are no $GCRX_{i,j}$; and similarly for GSY_i.

v) For each a between 1 (one) and *NX* and each b between 1 (one) and *NY*, let $GST_{a,b}$ be

```
( GCRXa,1, ..., GCRXa,MX(a), GCRYb,1 , ..., GCRYb,MY(b) )
```

that is, an <ordinary grouping set> consisting of $GCRA_{a,j}$ for all j between 1 (one) and $MX(a)$, followed by $GCRY_{b,j}$ for all j between 1 (one) and $MY(b)$.

vi) *CGB* is transformed by replacing "*GSSX* <comma> *GSSY*" with

```
GROUPING SETS (
  GST1,1, ..., GST1,NY,
  GST2,1, ..., GST2,NY,
  ...
  GSTNX,1 , ..., GSTNX,NY
)
```

NOTE 163 — Thus each <ordinary grouping set> in *GSSA* is "concatenated" with each <ordinary grouping set> in *GSSB*. For example,

```
GROUP BY GROUPING SETS ((A, B), (C)),
         GROUPING SETS ((X, Y), ())
```

is transformed to

```
GROUP BY GROUPING SETS ((A, B, X, Y), (A, B),
         (C, X, Y), (C))
```

d) The previous subrule of this Syntax Rule is applied repeatedly until *CGB* consists of a single <grouping sets specification>.

17) If <grouping element list> consists of a single <grouping sets specification> *GSS* that contains only <ordinary grouping set>s or <empty grouping set>s, then:

a) Let m be the number of <grouping set>s contained in *GSS*.

b) Let GS_i, $1 \le i \le m$, range over the <grouping set>s contained in *GSS*.

c) Let p be the number of distinct <column reference>s that are contained in *GSS*.

d) Let *PC* be an ordered list of these <column reference>s ordered according to their left-to-right occurrence in the list.

e) Let PC_k, $1 \le k \le p$, be the k-th <column reference> in *PC*.

f) Let $DTPC_k$ be the declared type of the column identified by PC_k.

g) Let *NDC* be the number of <derived column>s simply contained in *SL1*.

h) Let DC_q, $1 \le q \le NDC$, be an enumeration of the <derived column>s simply contained in SL1, in order from left to right.

i) Let DCN_q be the column name of DC_q, 1 (one) $\le q \le NDC$.

j) Let VE_q, 1 (one) $\le q \le NDC$, be the <value expression> simply contained in DC_q.

k) Let XN_k, 1 (one) $\le k \le p$, YN_k, 1 (one) $\le k \le p$, and ZN_q, 1 (one) $\le q \le NDC$, be implementation-dependent column names that are all distinct from one another.

l) Let *SL2* be the <select list>:

```
PC1 AS XN1, GROUPING (PC1) AS YN1,
...,
PCp AS XNp, GROUPING (PCp) AS YNp,
VE1 AS XN1, ..., VENDC AS ZNNDC
```

m) For each GS_i:

i) If GS_i is an <empty grouping set>, then let $n(i)$ be 0 (zero). If GS_i is a <grouping column reference>, then let $n(i)$ be 1 (one). Otherwise, let $n(i)$ be the number of <grouping column reference>s contained in the <grouping column reference list>.

ii) Let $GCR_{i,j}$, $1 \le j \le n(i)$, range over the <grouping column reference>s contained in GS_i.

iii) Case:

1) If GS_i is an <ordinary grouping set>, then

A) Transform *SL2* to obtain *SL3*, and transform *HC* to obtain *HC3*, as follows:

For every PC_k, if there is no j such that $PC_k = GCR_{i,j}$, then make the following replacements in *SL2* and *HC*:

I) Replace each <grouping operation> in *SL2* and *HC* that contains a <column reference> that references PC_k by the <literal> 1 (one).

II) Replace each <column reference> in *SL2* and *HC* that references PC_k by

```
CAST ( NULL AS DTPCk )
```

B) Transform *SL3* to obtain *SLNEW*, and transform *HC3* to obtain *HCNEW* by replacing each <grouping operation> that remains in *SL3* and *HC3* by the <literal> 0 (zero).

NOTE 164 — Thus the value of a <grouping operation> is 0 (zero) if the grouping column referenced by the <grouping operation> is among the $GCR_{i,j}$ and 1(one) if it is not.

C) Let $GSSQL_i$ be:

```
SELECT QSSQ SLNEW
FC
WC
GROUP BY GCRi,1, ..., GCRi,n(i)
HCNEW
```

2) If GS_i is an <empty grouping set>, then

A) Transform *SL2* to obtain *SLNEW*, and transform *HC* to obtain *HCNEW*, as follows:

For every k, $1 \le k \le p$:

I) Replace each <grouping operation> in *SL2* and *HC* that contains a <column reference> that references $PC_k$by the <literal> 1 (one).

II) Replace each <column reference> in *SL2* and *HC* that references PC_k by

```
CAST ( NULL AS DTPC_k )
```

B) Let $GSSQL_i$ be

```
SELECT QSSQ SLNEW
FC
WC
GROUP BY ()
HCNEW
```

n) Let *GU* be:

```
GSSQL_1
  UNION GBSQ
GSSQL_2
  UNION GBSQ
...
  UNION GBSQ
GSSQL_m
```

o) *QS* is equivalent to

```
SELECT QSSQ ZN_1 AS DC_1, ..., ZN_NDC AS DC_NDC
FROM ( GU )
```

Access Rules

None.

General Rules

NOTE 165 — As a result of the syntactic transformations specified in the Syntax Rules of this Subclause, only primitive <group by clause>s are left to consider.

1) If no <where clause> is specified, then let *T* be the result of the preceding <from clause>; otherwise, let *T* be the result of the preceding <where clause>.

2) Case:

   a) If there are no grouping columns, then the result of the <group by clause> is the grouped table consisting of *T* as its only group.

   b) Otherwise, the result of the <group by clause> is a partitioning of the rows of *T* into the minimum number of groups such that, for each grouping column of each group, no two values of that grouping column are distinct. If the declared type of a grouping column is a user-defined type and the comparison of that column results in *Unknown* for two rows of *T*, then the assignment of those rows to groups in the result of the <group by clause> is implementation-dependent.

3) When a <search condition> or <value expression> is applied to a group, a reference *CR* to a column that is functionally dependent on the grouping columns is understood as follows.

   Case:

a) If *CR* is a group-invariant column reference, then it is a reference to the common value in that column of the rows in that group. If the most specific type of the column is character, datetime with time zone, or a user-defined type, then the value is an implementation-dependent value that is not distinct from the value of the column in each row of the group.

b) Otherwise, *CR* is a within-group-varying column reference, and as such, it is a reference to the value of the column in each row of a given group determined by the grouping columns, to be used to construct the argument source of a <set function specification>.

Conformance Rules

1) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain a <rollup list>.

2) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain a <cube list>.

3) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain a <grouping sets specification>.

4) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain an <empty grouping set>.

5) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain an <ordinary grouping set> that contains a <grouping column reference list>.

6) Without Feature T432, “Nested and concatenated GROUPING SETS”, conforming SQL language shall not contain a <grouping set list> that contains a <grouping sets specification>.

7) Without Feature T432, “Nested and concatenated GROUPING SETS”, conforming SQL language shall not contain a <group by clause> that simply contains a <grouping sets specification> *GSS* where *GSS* is not the only <grouping element> simply contained in the <group by clause>.

   NOTE 166 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

8) Without Feature T434, “GROUP BY DISTINCT”, conforming SQL language shall not contain a <group by clause> that simply contains a <set quantifier>.

7.10 <having clause>

Function

Specify a grouped table derived by the elimination of groups that do not satisfy a <search condition>.

Format

```
<having clause> ::=
  HAVING <search condition>
```

Syntax Rules

1) Let *HC* be the <having clause>. Let *TE* be the <table expression> that immediately contains *HC*. If *TE* does not immediately contain a <group by clause>, then "GROUP BY ()" is implicit. Let *T* be the descriptor of the table defined by the <group by clause> *GBC* immediately contained in *TE* and let *R* be the result of *GBC*.

2) Let *G* be the set consisting of every column referenced by a <column reference> contained in *GBC*.

3) Each column reference directly contained in the <search condition> shall be one of the following:

   a) An unambiguous reference to a column that is functionally dependent on *G*.

   b) An outer reference.

   NOTE 167 — See also the Syntax Rules of Subclause 6.7, "<column reference>".

4) The <search condition> shall not contain a <window function> without an intervening <query expression>.

5) The row type of the result of the <having clause> is the row type *RT* of *T*.

Access Rules

None.

General Rules

1) The <search condition> is evaluated for each group of *R*. The result of the <having clause> is a grouped table of those groups of *R* for which the result of the <search condition> is *True*.

Conformance Rules

1) Without Feature T301, "Functional dependencies", in conforming SQL language, each column reference directly contained in the <search condition> shall be one of the following:

   a) An unambiguous reference to a grouping column of *T*.

   b) An outer reference.

2) Without Feature T301, “Functional dependencies”, in conforming SQL language, each column reference contained in a <query expression> in the <search condition> that references a column of *T* shall be one of the following:

   a) An unambiguous reference to a grouping column of *T*.

   b) Contained in an aggregated argument of a <set function specification>.

7.11 <window clause>

Function

Specify one or more window definitions.

Format

```
<window clause> ::=
  WINDOW <window definition list>

<window definition list> ::=
  <window definition> [ { <comma> <window definition> }... ]

<window definition> ::=
  <new window name> AS <window specification>

<new window name> ::=
  <window name>

<window specification> ::=
  <left paren> <window specification details> <right paren>

<window specification details> ::=
  [ <existing window name> ]
      [ <window partition clause> ]
      [ <window order clause> ]
      [ <window frame clause> ]

<existing window name> ::=
  <window name>

<window partition clause> ::=
  PARTITION BY <window partition column reference list>

<window partition column reference list> ::=
  <window partition column reference>
      [ { <comma> <window partition column reference> }... ]

<window partition column reference> ::=
  <column reference> [ <collate clause> ]

<window order clause> ::=
  ORDER BY <sort specification list>

<window frame clause> ::=
  <window frame units> <window frame extent>
      [ <window frame exclusion> ]

<window frame units> ::=
    ROWS
  | RANGE

<window frame extent> ::=
    <window frame start>
  | <window frame between>
```

```
<window frame start> ::=
    UNBOUNDED PRECEDING
  | <window frame preceding>
  | CURRENT ROW

<window frame preceding> ::=
  <unsigned value specification> PRECEDING

<window frame between> ::=
  BETWEEN <window frame bound 1> AND <window frame bound 2>

<window frame bound 1> ::=
  <window frame bound>

<window frame bound 2> ::=
  <window frame bound>

<window frame bound> ::=
    <window frame start>
  | UNBOUNDED FOLLOWING
  | <window frame following>

<window frame following> ::=
  <unsigned value specification> FOLLOWING

<window frame exclusion> ::=
    EXCLUDE CURRENT ROW
  | EXCLUDE GROUP
  | EXCLUDE TIES
  | EXCLUDE NO OTHERS
```

Syntax Rules

1) Let *TE* be the <table expression> that immediately contains the <window clause>.

2) <new window name> *NWN1* shall not be contained in the scope of another <new window name> *NWN2* such that *NWN1* and *NWN2* are equivalent.

3) Let *WDEF* be a <window definition>.

4) Each <column reference> contained in the <window partition clause> or <window order clause> of *WDEF* shall unambiguously reference a column of the derived table *T* that is the result of *TE*. A column referenced in a <window partition clause> is a *partitioning column*. Each partitioning column is an operand of a grouping operation, and the Syntax Rules of Subclause 9.10, "Grouping operations", apply.

   NOTE 168 — If *T* is a grouped table, then the <column reference>s contained in <window partition clause> or <window order clause> shall reference columns of the grouped table obtained by performing the syntactic transformation in Subclause 7.12, "<query specification>".

5) For every <window partition column reference> *PC*,

   Case:

   a) If <collate clause> is specified, then let *CS* be the collation identified by <collation name>. The declared type of the column reference shall be character string. The declared type of *PC* is that of its column reference, except that *CS* is the declared type collation and the collation derivation is *explicit*.

   b) Otherwise, the declared type of *PC* is the declared type of its column reference.

6) If *T* is a grouped table, then let *G* be the set of grouping columns of *T*. Each column reference contained in <window clause> that references a column of *T* shall reference a column that is functionally dependent on *G* or be contained in an aggregated argument of a <set function specification>.

7) A <window clause> shall not contain a <window function> without an intervening <query expression>.

8) If *WDEF* specifies <window frame between>, then:

   a) <window frame bound 1> shall not specify UNBOUNDED FOLLOWING.

   b) <window frame bound 2> shall not specify UNBOUNDED PRECEDING.

   c) If <window frame bound 1> specifies CURRENT ROW, then <window frame bound 2> shall not specify <window frame preceding>.

   d) If <window frame bound 1> specifies <window frame following>, then <window frame bound 2> shall not specify <window frame preceding> or CURRENT ROW.

9) If *WDEF* specifies <window frame extent>, and does not specify <window frame between>, then let *WAGS* be the <window frame start>. The <window frame extent> is equivalent to

```
BETWEEN WAGS AND CURRENT ROW
```

10) If *WDEF* specifies an <existing window name> *EWN*, then:

   a) *WDEF* shall be within the scope of a <window name> that is equivalent to <existing window name>.

   b) Let *WDX* be the window structure descriptor identified by *EWN*.

   c) *WDEF* shall not specify <window partition clause>.

   d) If *WDX* has a window ordering clause, then *WDEF* shall not specify <window order clause>.

   e) *WDX* shall not have a window framing clause.

11) If *WDEF*'s <window frame clause> specifies <window frame preceding> or <window frame following>, then let *UVS* be the <unsigned value specification> simply contained in the <window frame preceding> or <window frame following>.

   Case:

   a) If RANGE is specified, then *WDEF*'s <window order clause> shall contain a single <sort key> *SK*. The declared type of *SK* shall be numeric, datetime, or interval. The declared type of *UVS* shall be numeric if the declared type of *SK* is numeric; otherwise, it shall be an interval type that may be added to or subtracted from the declared type of *SK* according to the Syntax Rules of Subclause 6.30, "<datetime value expression>", and Subclause 6.32, "<interval value expression>", in this part of ISO/IEC 9075.

   b) If ROWS is specified, then the declared type of *UVS* shall be exact numeric with scale 0 (zero).

12) The scope of the <new window name> simply contained in *WDEF* consists of any <window definition>s that follow *WDEF* in the <window clause>, together with the <select list> of the <query specification> or <select statement: single row> that simply contains the <window clause>. If the <window clause> is simply contained in a <query specification> that is the <query expression body> of a <cursor specification> that is a simple table query, then the scope of <new window name> also includes the <order by clause> of the <cursor specification>.

13) Two window structure descriptors *WD1* and *WD2* are *order-equivalent* if all of the following conditions are met:

a) Let $WPCR1_i$, 1 (one) $\le i \le N1$, and $WPCR2_i$, 1 (one) $\le i \le N2$, be enumerations of the <window partition column reference>s contained in the window partitioning clauses of *WD1* and *WD2*, respectively, in order from left to right. $N1 = N2$, and, for all *i*, $WPCR1_i$ and $WPCR2_i$ are equivalent column references.

b) Let $SS1_i$, 1 (one) $\le i \le M1$, and $SS2_i$, 1 (one) $\le i \le M2$, be enumerations of the <sort specification>s contained in the window ordering clauses of *WD1* and *WD2*, respectively, in order from left to right. $M1 = M2$, and, for all *i*, $SS1_i$ and $SS2_i$ contain <sort key>s that are equivalent column references, specify or imply the same <ordering specification>, specify or imply the same <collate clause>, if any, and specify or imply the same <null ordering>.

Access Rules

None.

General Rules

1) Let *TE* be the <table expression> that simply contains the <window clause>. Let *SL* be the <select list> of the <query specification> or <select statement: single row> that immediately contains *TE*.

Case:

a) If *SL* does not simply contain a <window function>, then the <window clause> is disregarded, and the result of *TE* is the result of the last <from clause>, <where clause>, <group by clause> or <having clause> of *TE*.

b) Otherwise, let *RTE* be the result of the last <from clause> or <where clause> simply contained in *TE*.

NOTE 169 — Although it is permissible to have a <group by clause> or a <having clause> with a <window clause>, if there are any <window function>s, then the <group by clause> and <having clause> are removed by a syntactic transformation in Subclause 7.12, "<query specification>", and so are not considered here.

i) A window structure descriptor *WDESC* is created for each <window definition> *WDEF*, as follows:

1) *WDESC*'s window name is the <new window name> simply contained in *WDEF*.

2) If <existing window name> is specified, then let *EWN* be the <existing window name> simply contained in *WDEF* and let *WDX* be the window structure descriptor identified by *EWN*.

3) If <existing window name> is specified and the window ordering clause of *WDX* is present, then the ordering window name of *WDESC* is *EWN*; otherwise, there is no ordering window name.

4) Case:

A) If *WDEF* simply contains <window partition clause> *WDEFWPC*, then *WDESC*'s window partitioning clause is *WDEFWPC*.

B) If <existing window name> is specified, then *WDESC*'s window partitioning clause is the window partitioning clause of *WDX*.

C) Otherwise, *WDESC* has no window partitioning.

5) Case:

A) If *WDEF* simply contains <window order clause> *WDEFWOC*, then *WDESC*'s window ordering clause is *WDEFWOC*.

B) If <existing window name> is specified, then *WDESC*'s window ordering clause is the window ordering clause of *WDX*.

C) Otherwise, *WDESC* has no window ordering.

6) If *WDEF* simply contains <window frame clause> *WDEFWFC*, then *WDESC*'s window framing clause is *WDEFWFC*; otherwise, *WDESC* has no windows framing.

ii) The result of <window clause> is *RTE*, together with the window structure descriptors defined by the <window clause>.

2) Let *WD* be a window structure descriptor.

3) *WD* defines, for each row *R* of *RTE*, the *window partition* of *R* under *WD*, consisting of the collection of rows of *RTE* that are not distinct from *R* in the window partitioning columns of *WD*. If *WD* has no window partitioning clause, then the window partition of *R* is the entire result *RTE*.

4) *WD* also defines the window ordering of the rows of each window partition defined by *WD*, according to the General Rules of Subclause 10.10, "<sort specification list>", using the <sort specification list> simply contained in *WD*'s window ordering clause. If *WD* has no window ordering clause, then the window ordering is implementation-dependent, and all rows are peers. Although the window ordering of peer rows within a window partition is implementation-dependent, the window ordering shall be the same for all window structure descriptors that are order-equivalent. It shall also be the same for any pair of windows *W1* and *W2* such that *W1* is the ordering window for *W2*.

5) *WD* also defines for each row *R* of *RTE* the window frame *WF* of *R*, consisting of a collection of rows. *WF* is defined as follows.

Case:

a) If *WD* has no window framing clause, then

Case:

i) If the window ordering clause of *WD* is not present, then *WF* is the window partition of *R*.

ii) Otherwise, *WF* consists of all rows of the partition of *R* that precede *R* or are peers of *R* in the window ordering of the window partition defined by the window ordering clause.

b) Otherwise, let *WF* initially be the window partition of *R* defined by *WD*. Let *WFC* be the window framing clause of *WD*. Let *WFB1* be the <window frame bound 1> and let *WFB2* be the <window frame bound 2> contained in *WFC*.

i) If RANGE is specified, then:

1) In the following subrules, when performing addition or subtraction to combine a datetime and a year-month interval, if the result would raise the exception condition *data exception — datetime field overflow* because the <primary datetime field> DAY is not valid for the computer value of the <primary datetime field>s YEAR and MONTH, then the <primary

datetime field> DAY is set to the last day that is valid for the <primary datetime field>s YEAR and MONTH, and no exception condition is raised.

2) Case:

NOTE 170 — In the following subrules, if *WFB1* specifies UNBOUNDED PRECEDING, then no rows are removed from *WF* by this step. *WFB1* may not be UNBOUNDED FOLLOWING.

A) If *WFB1* specifies <window frame preceding>, then let *V1P* be the value of the <unsigned value specification>.

Case:

I) If *V1P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

a) If NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

b) Case:

i) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK+V1P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

ii) Otherwise, let *BOUND* be the value *VSK–V1P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

B) If *WFB1* specifies CURRENT ROW, then remove from *WF* all rows that are not peers of the current row and that precede the current row in the window ordering defined by *WD*.

C) If *WFB1* specifies <window frame following>, then let *V1F* be the value of the <unsigned value specification>.

Case:

I) If *V1F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

a) If NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

b) Case:

i) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK–V1F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

ii) Otherwise, let *BOUND* be the value *VSK+V1F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

3) Case:

NOTE 171 — In the following subrules, if *WFB2* specifies UNBOUNDED FOLLOWING, then no rows are removed from *WF* by this step. *WFB2* may not be UNBOUNDED PRECEDING.

A) If *WFB2* specifies <window frame preceding>, then let *V2P* be the value of the <unsigned value specification>.

Case:

I) If *V2P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

a) If NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

b) Case:

i) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK+V2P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

ii) Otherwise, let *BOUND* be the value *VSK–V2P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

B) If *WFB2* specifies CURRENT ROW, then remove from *WF* all rows following the current row in the ordering defined by *WD* that are not peers of the current row.

C) If *WFB2* specifies <window frame following>, then let *V2F* be the value of the <unsigned value specification>.

Case:

I) If *V2F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

1) If *VSK* is the null value and if NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

2) If *VSK* is not the null value, then:

a) If NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

b) Case:

i) If the <ordering specification> contained in the <window order clause> specifies DESC, then let *BOUND* be the value *VSK–V2F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

ii) Otherwise, let *BOUND* be the value *VSK+V2F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

ii) If ROWS is specified, then:

1) Case:

NOTE 172 — In the following subrules, if *WFB1* specifies UNBOUNDED PRECEDING, then no rows are removed from *WF* by this step. *WFB1* may not be UNBOUNDED FOLLOWING.

A) If *WFB1* specifies <window frame preceding>, then let *V1P* be the value of the <unsigned value specification>.

Case:

I) If *V1P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that are more than *V1P* rows preceding the current row in the window ordering defined by *WD*.

B) If *WFB1* specifies CURRENT ROW, then remove from *WF* all rows that precede the current row in the window ordering defined by *WD*.

NOTE 173 — This step removes any peers of the current row that precede it in the implementation-dependent window ordering.

C) If *WFB1* specifies <window frame following>, then let *V1F* be the value of the <unsigned value specification>.

Case:

I) If *V1F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that precede the current row and all rows that are less than *V1F* rows following the current row in the window ordering defined by *WD*.

NOTE 174 — If *V1F* is zero, then the current row is not removed from *WF* by this step; otherwise, the current row is removed from *WF*.

2) Case:

NOTE 175 — In the following subrules, if *WFB2* specifies UNBOUNDED FOLLOWING, then no rows are removed from *WF* by this step. *WFB2* may not be UNBOUNDED PRECEDING.

A) If *WFB2* specifies <window frame preceding>, then let *V2P* be the value of the <unsigned value specification>.

Case:

I) If *V2P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that follow the current row and all rows that are less than *V2P* rows preceding the current row in the window ordering defined by *WD*.

NOTE 176 — If *V2P* is zero, then the current row is not removed from *WF* by this step; otherwise, the current row is removed from *WF*.

B) If *WFB2* specifies CURRENT ROW, then remove from *WF* all rows that follow the current row in the window ordering defined by *WD*.

NOTE 177 — This step removes any peers of the current row that follow it in the implementation-dependent window ordering.

C) If *WFB2* specifies <window frame following>, then let *V2F* be the value of the <unsigned value specification>.

Case:

I) If *V2F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function*.

II) Otherwise, remove from *WF* all rows that are more than *V2F* rows following the current row in the window ordering defined by *WD*.

iii) If <window frame exclusion> *WFE* is specified, then

Case:

1) If EXCLUDE CURRENT ROW is specified and the current row is still a member of *WF*, then remove the current row from *WF*.

2) If EXCLUDE GROUP is specified, then remove the current row and any peers of the current row from *WF*.

3) If EXCLUDE TIES is specified, then remove any rows other than the current row that are peers of the current row from *WF*.

NOTE 178 — If the current row is already removed from *WF*, then it remains removed from *WF*.

NOTE 179 — If EXCLUDE NO OTHERS is specified, then no additional rows are removed from *WF* by this Rule.

Conformance Rules

1) Without Feature T611, “Elementary OLAP operations”, conforming SQL language shall not contain a <window specification>.

2) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window clause>.

3) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain an <existing window name>.

4) Without Feature T301, “Functional dependencies”, in conforming SQL language, if *T* is a grouped table, then each column reference contained in <window clause> that references a column of *T* shall be a reference to a grouping column of *T* or be contained in an aggregated argument of a <set function specification>.

5) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window frame exclusion>.

NOTE 180 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

7.12 <query specification>

This Subclause is modified by Subclause 7.1, "<query specification>", in ISO/IEC 9075-4.

Function

Specify a table derived from the result of a <table expression>.

Format

```
<query specification> ::=
  SELECT [ <set quantifier> ] <select list> <table expression>

<select list> ::=
    <asterisk>
  | <select sublist> [ { <comma> <select sublist> }... ]

<select sublist> ::=
    <derived column>
  | <qualified asterisk>

<qualified asterisk> ::=
    <asterisked identifier chain> <period> <asterisk>
  | <all fields reference>

<asterisked identifier chain> ::=
  <asterisked identifier> [ { <period> <asterisked identifier> }... ]

<asterisked identifier> ::=
  <identifier>

<derived column> ::=
  <value expression> [ <as clause> ]

<as clause> ::=
  [ AS ] <column name>

<all fields reference> ::=
  <value expression primary> <period> <asterisk>
      [ AS <left paren> <all fields column name list> <right paren> ]

<all fields column name list> ::=
  <column name list>
```

Syntax Rules

1) Let *T* be the result of the <table expression>.

2) Let *TQS* be the table that is the result of a <query specification>.

3) Case:

a) If the <select list> "*" is simply contained in a <table subquery> that is immediately contained in an <exists predicate>, then the <select list> is equivalent to a <value expression> that is an arbitrary <literal>.

b) Otherwise, the <select list> "*" is equivalent to a <value expression> sequence in which each <value expression> is a column reference that references a column of *T* and each column of *T* is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within *T*.

4) The degree of the table specified by a <query specification> is equal to the cardinality of the <select list>.

5) If a <set quantifier> DISTINCT is specified, then each column of *T* is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

6) The ambiguous case of an <all fields reference> whose <value expression primary> takes the form of an <asterisked identifier chain> shall be analyzed first as an <asterisked identifier chain> to resolve the ambiguity.

7) If <asterisked identifier chain> is specified, then:

a) Let *IC* be an <asterisked identifier chain>.

b) Let *N* be the number of <asterisked identifier>s immediately contained in *IC*.

c) Let I_i, 1 (one) ≤ i ≤ N, be the <asterisked identifier>s immediately contained in *IC*, in order from left to right.

d) Let PIC_1 be I_1. For each *J* between 2 and *N*, let PIC_J be $PIC_{J-1}.I_J$. PIC_J is called the *J*-th *partial identifier chain* of *IC*.

e) Let *M* be the minimum of *N* and 3.

f) For at most one *J* between 1 and *M*, PIC_J is called the *basis* of *IC*, and *J* is called the *basis length* of *IC*. The *referent* of the basis is a table *T*, a column *C* of a table, or an SQL parameter *SP*. The *basis* and *basis scope* of *IC* are defined in terms of a *candidate basis*, according to the following rules:

i) If *IC* is contained in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_1, then PIC_1 is a candidate basis of *IC*, and the scope of PIC_1 is the scope of *SP*.

ii) If $N = 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iii) If $N > 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes a refinable SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iv) If $N = 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. If the table associated with *EN* has a column *C* of row type whose <identifier> is equivalent to I_2, then PIC_2 is a candidate basis of *IC* and the scope of PIC_2 is the scope of *EN*.

v) If $N > 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. If the table associated with *EN* has a column *C* of row type or structured type whose <identifier> is equivalent to I_2, then PIC_2 is a candidate basis of *IC* and the scope of PIC_2 is the scope of *EN*.

vi) If $N \leq 3$ and PIC_N is equivalent to an exposed <table or query name> that is in scope, then let *EN* be the exposed <table or query name> that is equivalent to PIC_N and has the innermost scope. PIC_N is a candidate basis of *IC*, and the scope of PIC_N is the scope of *EN*.

vii) There shall be exactly one candidate basis *CB* with innermost scope. The basis of *IC* is *CB*. The basis scope is the scope of *CB*.

g) Case:

i) If the basis is a <table or query name> or <correlation name>, then let *TQ* be the table associated with the basis. The <select sublist> is equivalent to a <value expression> sequence in which each <value expression> is a column reference *CR* that references a column of *TQ* that is not a common column of a <joined table>. Each column of *TQ* that is not a referenced common column shall be referenced exactly once. The columns shall be referenced in the ascending sequence of their ordinal positions within *TQ*.

ii) Otherwise let *BL* be the length of the basis of *IC*.

Case:

1) If $BL = N$, then the <select sublist> `IC.*` is equivalent to `(IC).*`.

2) Otherwise, the <select sublist> `IC.*` is equivalent to:

$$(PIC_{BL}) . I_{BL+1} . \ldots . I_N . *$$

NOTE 181 — The equivalent syntax in either case will be analyzed as <all fields reference> ::= <value expression primary> <period> <asterisk>

8) The data type of the <value expression primary> *VEP* specified in an <all fields reference> *AFR* shall be some row type *VER*. Let *n* be the degree of *VER*. Let $F_1, \ldots, F_N$ be the field names of *VER*.

Case:

a) If <all fields column name list> *AFCNL* is specified, then the number of <column name>s simply contained in *AFCNL* shall be *n*. Let $AFCN_i$, 1 (one) $\leq i \leq n$, be these <column name>s in order from left to right. *AFR* is equivalent to

$$VEP . F_1 \text{ AS } AFCN_1, \ldots , VEP . F_n \text{ AS } AFCN_n)$$

b) Otherwise, *AFR* is equivalent to:

$$VEP . F_1 , \ldots , VEP . F_n$$

9) Let *C* be some column. Let *QS* be the <query specification>. Let DC_i, for *i* ranging from 1 (one) to the number of <derived column>s inclusively, be the *i*-th <derived column> simply contained in the <select list> of *QS*. For all *i*, *C* is an underlying column of DC_i, and of any column reference that identifies DC_i, if and only if *C* is an underlying column of the <value expression> of DC_i, or *C* is an underlying column of the <table expression> immediately contained in *QS*.

10) Each column reference contained in a <window function> shall unambiguously reference a column of *T*.

11) If both of the following two conditions are satisfied, then *QS* is a *grouped, windowed query*:

    a) *T* is a grouped table.

    b) Some <derived column> simply contained in *QS* simply contains a <window function>.

12) A grouped, windowed query *GWQ* is transformed to an equivalent <query specification> as follows:

    a) If *GWQ* contains an <in-line window specification>, then apply the syntactic transformation specified in Subclause 6.10, "<window function>".

    b) If the <select list> of *GWQ* immediately contains <asterisk> or simply contains <qualified asterisk>, then apply the syntactic transformations specified in Subclause 7.12, "<query specification>".

    c) Let *GWQ2* be the result of the preceding transformations, if any.

    d) Let *SL*, *FC*, *WC*, *GBC*, *HC*, and *WIC* be the <select list>, <from clause>, <where clause>, <group by clause>, <having clause>, and <window clause>, respectively, of *GWQ2*. If any of <where clause>, <group by clause>, or <having clause>, are missing, then let *WC*, *GBC*, and *HC*, respectively, be a zero-length string. Let *SQ* be the <set quantifier> immediately contained in the <query specification> of *GWQ2*, if any; otherwise, let *SQ* be a zero-length string.

       NOTE 182 — *GWQ2* can not lack a <window clause>, since the syntactic transformation of Subclause 6.10, "<window function>", will create one if there is not one in *GWQ* already.

    e) Let *N1* be the number of <set function specification>s simply contained in *GWQ2*.

    f) Let SFS_i, 1 (one) ≤ *i* ≤ *N1*, be an enumeration of the <set function specification>s simply contained in *GWQ2*.

    g) Let $SFSI_i$, 1 (one) ≤ *i* ≤ *N1*, be a list of <identifier>s that are distinct from each other and distinct from all <identifier>s contained in *GWQ2*.

    h) If *N1* = 0 (zero), then let *SFSL* be a zero-length string; otherwise, let *SFSL* be:

       SFS_1 AS $SFSI_1$,
       SFS_2 AS $SFSI_2$, ...,
       SFS_{N1} AS $SFSI_{N1}$

    i) Let *HCNEW* be obtained from *HC* by replacing each <set function specification> SFS_i by the corresponding <identifier> $SFSI_i$.

    j) Let *N2* be the number of <column reference>s that are contained in *SL* or *WIC* without an intervening <query expression> or <set function specification>.

    k) Let CR_j, 1 (one) ≤ *j* ≤ *N2*, be an enumeration of the <column reference>s that are contained in *SL* or *WIC* without an intervening <query expression> or <set function specification>.

l) Let CRI_j, 1 (one) ≤ j ≤ $N2$, be a list of <identifier>s that are distinct from each other, distinct from all identifiers in *GWQ2*, and distinct from all $SFSI_i$.

m) If $N2$ = 0 (zero), then let *CRL* be a zero-length string; otherwise, let *CRL* be:

```
CR1 AS CRI1, CR2 AS CRI2, ..., CRN2 AS CRIN2
```

n) Let *N3* be the number of <derived column>s simply contained in *SL* that do not specify <as clause>.

o) Let $DCOL_k$, 1 (one) ≤ k ≤ $N3$, be the <derived column>s simply contained in *SL* that do not specify an <as clause>. For each k, let $COLN_k$ be the <column name> determined as follows.

Case:

i) If $DCOL_k$ is a single column reference, then let $COLN_k$ be the <column name> of the column designated by the column reference.

ii) Otherwise, let $COLN_k$ be an implementation-dependent <column name>.

p) Let *SL2* be obtained from *SL* by replacing each <derived column> $DCOL_k$ by

```
DCOLk AS COLNk
```

q) Let *GWQN* be an arbitrary <identifier>.

r) Let *SLNEW* be the <select list> obtained from *SL2* by replacing each simply contained <set function specification> SFS_i by $GWQN.SFSI_i$ and replacing each <column reference> CR_j that is contained without an intervening <query expression> or <set function specification> by $GWQN.CRI_j$.

s) Let *WICNEW* be the <window clause> obtained from *WIC* by replacing each <set function specification> SFS_i by $GWQN.SFSI_i$ and by replacing each <column reference> CR_j by $GWQN.CRI_j$.

t) If either *SFSL* or *CRL* is a zero-length string, then let *COMMA* be a zero-length string; otherwise, let *COMMA* be "," (a <comma>).

u) *GWQ* is equivalent to the following <query specification>:

```
SELECT SLNEW
FROM ( SELECT SQ SFSL COMMA CRL
       FC
       WC
       GBC
       HC ) AS GWQN
WICNEW
```

13) A <query specification> is *possibly non-deterministic* if any of the following conditions are true:

a) The <set quantifier> DISTINCT is specified and one of the columns of *T* has a data type of character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE.

b) The <query specification> generally contains a <value expression>, <query specification>, or <query expression> that is possibly non-deterministic.

c) The <select list>, <having clause>, or <window clause> contains a reference to a column *C* of *T* that has a data type of character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE, and the functional dependency *G* *C*, where *G* is the set consisting of the grouping columns of *T*, holds in *T*.

14) If <table expression> does not immediately contain a <group by clause> and <table expression> is simply contained in a <query expression> that is the aggregation query of some <set function specification>, then `GROUP BY ()` is implicit.

NOTE 183 — "aggregation query" is defined in Subclause 6.9, "<set function specification>".

15) If *T* is a grouped table, then let *G* be the set of grouping columns of *T*. In each <value expression> contained in <select list> , each column reference that references a column of *T* shall reference some column *C* that is functionally dependent on *G* or shall be contained in an aggregated argument of a <set function specification> whose aggregation query is *QS*.

NOTE 184 — See also the Syntax Rules of Subclause 6.7, "<column reference>".

16) Each column of *TQS* has a column descriptor that includes a data type descriptor that is the same as the data type descriptor of the <value expression> simply contained in the <derived column> defining that column.

17) Case:

a) If the *i*-th <derived column> in the <select list> specifies an <as clause> that contains a <column name> *CN*, then the <column name> of the *i*-th column of the result is *CN*.

b) If the *i*-th <derived column> in the <select list> does not specify an <as clause> and the <value expression> of that <derived column> is a single column reference, then the <column name> of the *i*-th column of the result is the <column name> of the column designated by the column reference.

c) Otherwise, the <column name> of the *i*-th column of the <query specification> is implementation-dependent.

18) A column of *TQS* is *known not null* if and only if at least one of the following conditions applies:

a) It is not defined by a <derived column> containing any of the following:

i) A column reference for a column *C* that is possibly nullable.

ii) An <indicator parameter>.

iii) An <indicator variable>.

iv) A <dynamic parameter specification>.

v) An SQL parameter.

vi) A <routine invocation>, <method reference>, or <method invocation> whose subject routine is an SQL-invoked routine that either is an SQL routine or is an external routine that specifies or implies PARAMETER STYLE SQL.

vii) A <query expression>.

viii) CAST (NULL AS *X*) (where *X* represents a <data type> or a <domain name>).

ix) A <window function> whose <window function type> does not contain <rank function type>, ROW_NUMBER, or an <aggregate function> that simply contains COUNT.

x) CURRENT_USER, CURRENT_ROLE, or SYSTEM_USER.

xi) A <set function specification> that does not contain COUNT.

xii) A <case expression>.

xiii) A <field reference>.

xiv) An <array element reference>.

xv) A <multiset element reference>.

xvi) A <dereference operation>.

xvii) A <reference resolution>.

xviii) A <comparison predicate>, <between predicate>, <in predicate>, or <quantified comparison predicate> *P* such that the declared type of a field of a <row value predicand> that is simply contained in *P* is a row type, a user-defined type, an array type, or a multiset type.

xix) A <member predicate>.

xx) A <submultiset predicate>.

b) An implementation-defined rule by which the SQL-implementation can correctly deduce that the value of the column cannot be null.

19) Let *TREF* be the <table reference>s that are simply contained in the <from clause> of the <table expression>. The *simply underlying table*s of the <query specification> are the <table or query name>s and <derived table>s contained in *TREF* without an intervening <derived table>.

20) The terms *key-preserving* and *one-to-one* are defined as follows:

a) Let *UT* denote some simply underlying table of *QS*, let *UTCOLS* be the set of columns of *UT*, let *QSCOLS* be the set of columns of *QS*, and let *QSCN* be an exposed range variable for *UT* whose scope clause is *QS*.

b) *QS* is said to be *key-preserving with respect to UT* if there is some strong candidate key *CKUT* of *UT* such that every member of *CKUT* has some counterpart under *QSCN* in *QSCOLS*.

NOTE 185 — "strong candidate key" is defined in Subclause 4.19, "Candidate keys".

NOTE 186 — "Counterpart" is defined in Subclause 4.18.2, "General rules and definitions". It follows from this condition that every row in *QS* corresponds to exactly one row in *UT*, namely that row in *UT* that has the same combined value in the columns of *CKUT* as the row in *QS*. There may be more than one row in *QS* that corresponds to a single row in *UT*.

c) *QS* is said to be *one-to-one with respect to UT* if and only if *QS* is key-preserving with respect to *UT*, *UT* is updatable, and there is some strong candidate key *CKQS* of *QS* such that every member of *CKQS* is a counterpart under *UT* of some member of *UTCOLS*.

NOTE 187 — It follows from this condition that every row in *UT* corresponds to at most one row in *QS*, namely that row in *QS* that has the same combined value in the columns of *CKQS* as the row in *UT*.

21) A <query specification> is *potentially updatable* if and only if the following conditions hold:

a) DISTINCT is not specified.

b) Of those <derived column>s in the <select list> that are column references that have a counterpart in a base table, no column of a table table is referenced more than once in the <select list>.

c) The <table expression> immediately contained in *QS* does not simply contain an explicit or implicit <group by clause> or a <having clause>.

22) If a <query specification> *QS* is potentially updatable, then

Case:

a) If the <from clause> of the <table expression> specifies exactly one <table reference>, then a column of *QS* is said to be a *potentially updatable column* if it has a counterpart in *TR* that is updatable.

NOTE 188 — The notion of updatable columns of table references is defined in Subclause 7.6, "<table reference>".

b) Otherwise, a column of *QS* is said to be a *potentially updatable column* if it has a counterpart in some updatable column of some simply underlying table *UT* of *QS* such that *QS* is one-to-one with respect to *UT*.

23) A <query specification> is *updatable* if it is potentially updatable and it has at least one potentially updatable column.

24) A <query specification> *QS* is *simply updatable* if the following conditions hold:

a) *QS* is updatable.

b) The <from clause> immediately contained in the <table expression> immediately contained in *QS* contains exactly one <table reference>, and the table referenced by that <table reference> is simply updatable.

c) Every result column of *QS* is updatable.

d) If the <table expression> immediately contained in *QS* immediately contains a <where clause> *WC*, then no leaf generally underlying table of *QS* is a generally underlying table of any <query expression> contained in *WC*.

25) A <query specification> *QS* is *insertable-into* if and only if every simply underlying table of *QS* is insertable-into.

26) A column *C* of *QS* is updatable if at least one of the following is true:

a) *QS* is simply updatable.

b) *QS* is updatable, *C* is potentially updatable, and the SQL implementation supports Feature T111, "Updatable joins, unions, and columns".

27) The row type *RT* of *TQS* is defined by the sequence of (<field name>, <data type>) pairs indicated by the sequence of column descriptors of *TQS* taken in order.

Access Rules

None.

General Rules

1) If *QS* is contained in a <subquery> *SQ*, then certain <set function specification>s and outer references are resolved, such that their values are constant for every row in the result of *QS*, as follows.

Case:

a) If *SQ* is being evaluated for a group, then let *G* be that group.

i) Let *TE* be the <table expression> whose result included *G*.

ii) For every <set function specification> *SFS* contained in *QS* whose aggregation query simply contains *TE*, the value of *SFS* is the result of evaluating *SFS* for *G*.

NOTE 189 — The circumstances in which a <subquery> is evaluated for a given group, rather than a given row, are defined in the General Rules of this Subclause and the General Rules of Subclause 7.10, "<having clause>".

b) Otherwise, let *R* be the row for which *SQ* is being evaluated. For every <column reference> *CR* contained in *SQ* that is an outer reference whose qualifying scope is simply contained in a <subquery> that contains *SQ*, the value of *CR* is the value of the field in *R* corresponding to the column referenced by *CR*.

NOTE 190 — An expression having been resolved under this rule is not resolved again in the case where it is contained in a <query expression> contained in *SQ*.

2) Case:

a) If *T* is not a grouped table, then each <value expression> is applied to each row of *T* yielding a table *TEMP* of *M* rows, where *M* is the cardinality of *T*. The *i*-th column of the table contains the values derived by the evaluation of the *i*-th <value expression>.

b) If *T* is a grouped table, then

Case:

i) If *T* has 0 (zero) groups, then let *TEMP* be an empty table.

ii) If *T* has one or more groups, then each <value expression> is applied to each group of *T* yielding a table *TEMP* of *M* rows, where *M* is the number of groups in *T*. The *i*-th column of *TEMP* contains the values derived by the evaluation of the *i*-th <value expression>. When a <value expression> is applied to a given group of *T*, that group is the argument source of each <set function specification> in the <value expression>.

3) Case:

a) If the <set quantifier> DISTINCT is not specified, then the result of the <query specification> is *TEMP*.

b) If the <set quantifier> DISTINCT is specified, then the result of the <query specification> is the table derived from *TEMP* by the elimination of all redundant duplicate rows. If the most specific type of any column is character string, datetime with time zone, or a user-defined type, then the precise values in those columns are chosen in an implementation-dependent fashion.

Conformance Rules

1) Without Feature F801, "Full set function", conforming SQL language shall not contain a <query specification> *QS* that contains more than one <set quantifier> containing DISTINCT, unless such <set quantifier> is contained in a <query expression> contained in *QS*.

2) Without Feature T051, "Row types", conforming SQL language shall not contain an <all fields reference>.

3) Without Feature T301, "Functional dependencies", in conforming SQL language, if *T* is a grouped table, then in each <value expression> contained in the <select list>, each <column reference> that references a column of *T* shall reference a grouping column or be specified in an aggregated argument of a <set function specification>.

4) Without Feature T325, “Qualified SQL parameter references”, conforming SQL language shall not contain an <asterisked identifier chain> whose referent is an SQL parameter and whose first <identifier> is the <qualified identifier> of a <routine name>.

5) Without Feature T053, “Explicit aliases for all-fields reference”, conforming SQL language shall not contain an <all fields column name list>.

NOTE 191 — If a <set quantifier> DISTINCT is specified, then the Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

7.13 <query expression>

This Subclause is modified by Subclause 7.2, "<query expression>", in ISO/IEC 9075-14.

Function

Specify a table.

Format

```
<query expression> ::=
  [ <with clause> ] <query expression body>

<with clause> ::=
  WITH [ RECURSIVE ] <with list>

<with list> ::=
  <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
  <query name> [ <left paren> <with column list> <right paren> ]
      AS <left paren> <query expression> <right paren> [ <search or cycle clause> ]

<with column list> ::=
  <column name list>

<query expression body> ::=
    <query term>
  | <query expression body> UNION [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>
  | <query expression body> EXCEPT [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>

<query term> ::=
    <query primary>
  | <query term> INTERSECT [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query primary>

<query primary> ::=
    <simple table>
  | <left paren> <query expression body> <right paren>

<simple table> ::=
    <query specification>
  | <table value constructor>
  | <explicit table>

<explicit table> ::=
  TABLE <table or query name>

<corresponding spec> ::=
  CORRESPONDING [ BY <left paren> <corresponding column list> <right paren> ]

<corresponding column list> ::=
  <column name list>
```

Syntax Rules

1) Let *QE* be the <query expression>.

2) If <with clause> is specified, then:

   a) If a <with clause> *WC* immediately contains RECURSIVE, then *WC*, its <with list>, and its <with list element>s are said to be *potentially recursive*. Otherwise they are said to be *non-recursive*.

   b) Let *n* be the number of <with list element>s. For each *i* , 1 (one) ≤ *i* < *n*, for each *j*, *i* < *j* ≤ *n*, the *j*-th <with list element> shall not immediately contain a <query name> that is equivalent to the <query name> immediately contained in the *i*-th <with list element>.

   c) If the <with clause> is non-recursive, then for all *i* between 1 (one) and *n*, the scope of the <query name> *WQN* immediately contained in the *i*-th <with list element> WLE_i is the <query expression> immediately contained in every <with list element> WLE_k, where *k* ranges from *i*+1 to *n*, and the <query expression body> immediately contained in <query expression>. A <table or query name> contained in this scope that immediately contains *WQN* is a *query name in scope*.

   d) If the <with clause> is potentially recursive, then for all *i* between 1 (one) and *n*, the scope of the <query name> *WQN* immediately contained in the *i*-th <with list element> WLE_i is the <query expression> immediately contained in every <with list element> WLE_k, where *k* ranges from 1 (one) to *n*, and the <query expression body> immediately contained in <query expression>. A <table or query name> contained in this scope that immediately contains *WQN* is a *query name in scope*.

   e) For every <with list element> *WLE*, let *WQE* be the <query expression> specified by *WLE* and let *WQT* be the table defined by *WQE*.

      i) If any two columns of *WQT* have equivalent names or if *WLE* is potentially recursive, then *WLE* shall specify a <with column list>. If *WLE* specifies a <with column list> *WCL*, then:

         1) Equivalent <column name>s shall not be specified more than once in *WCL*.

         2) The number of <column name>s in *WCL* shall be the same as the degree of *WQT*.

      ii) Every column of a character string type in *WQT* shall have a declared type collation.

   f) A *query name dependency graph QNDG* of a potentially recursive <with list> *WL* is a directed graph such that, for *i* ranging from 1 (one) to the number of <query name>s simply contained in *WL*:

      i) Each node represents a <query name> WQN_i immediately contained in a <with list element> WLE_i of *WL*.

      ii) Each arc from a node WQN_i to a node WQN_j represents the fact that WQN_j is referenced by a <query name> contained in the <query expression> immediately contained in WLE_i. WQN_i is said to *depend immediately* on WQN_j.

   g) For a potentially recursive <with list> *WL* with *n* elements, and for *i* ranging from 1 (one) to *n*, let WLE_i be the *i*-th <with list element> of *WL*, let WQN_i be the <query name> immediately contained in WLE_i, let WQE_i be the <query expression> immediately contained in WLE_i, let WQT_i be the table defined by WQE_i, and let *QNDG* be the query name dependency graph of *WL*.

      i) *WL* is said to be *recursive* if *QNDG* contains at least one cycle.

Case:

1) If *QNDG* contains an arc from WQN_i to itself, then WLE_i, WQN_i, and WQT_i are said to be *recursive*. WQN_i is said to belong to the *stratum* of WQE_i.

2) If *QNDG* contains a cycle comprising WQN_i, ..., WQN_k, with $k \neq i$, then it is said that WQN_i, ..., WQN_k are *recursive* and *mutually recursive* to each other, WQT_i, ..., WQT_k are *recursive* and *mutually recursive* to each other, and WLE_i, ..., WLE_k are *recursive* and *mutually recursive* to each other.

   For each *j* ranging from *i* to *k*, WQN_j belongs to the stratum of WQE_i,..., and WQE_k.

3) Among the WQE_i, ... WQE_k of a given stratum, there shall be at least one <query expression>, say WQE_j, such that:

   A) WQE_j is a <query expression body> that immediately contains UNION.

   B) WQE_j has one operand that does not contain a <query name> referencing any of WQN_i, ..., WQN_k. This operand is said to be the *non-recursive operand* of WQE_j.

   C) WQE_j is said to be an *anchor expression*, and WQN_j an *anchor name*.

   D) Let *CCCG* be the subgraph of *QNDG* that contains no nodes other than WQN_i, ..., WQN_k. For any anchor name WQN_j, remove the arcs to those query names WQN_l that are referenced by any <query name> contained in WQE_j. The remaining graph *SCCGP* shall not contain a cycle.

ii) If WLE_i is recursive, then

Case:

1) If WQE_i contains at most one WQN_k that belongs to the stratum of WQE_i, then WLE_i is *linearly recursive*.

2) Otherwise, let WQE_i contain any two <query name>s referencing WQN_k and WQN_l, both of which belong to the stratum of WQE_i.

   Case:

   A) WLE_i is *linearly recursive* if each of the following conditions is satisfied:

      I) WQE_i does not contain a <table reference list> that contains <query name>s referencing both WQN_k and WQN_l.

      II) WQE_i does not contain a <joined table> of which *TR1* and *TR2* are the first and second <table reference>s, or the <table reference> and <table factor>, respectively, and such that *TR1* and *TR2* contain <query name>s referencing WQN_k and WQN_l, respectively.

      III) WQE_i does not contain a <table expression> that immediately contains a <from clause> that contains WQN_k, and immediately contains a <where clause> containing a <query expression> that contains a <query name> referencing WQN_l.

B) Otherwise, WLE_i is said to be *non-linearly recursive*.

iii) For each WLE_i, for i ranging from 1 (one) to n, and for each WQN_j that belongs to the stratum of WQE_i:

1) WQE_i shall not contain a <query expression body> that contains a <query name> referencing WQN_j and immediately contains EXCEPT where the right operand of EXCEPT contains WQN_j.

2) WQE_i shall not contain a <routine invocation> with an <SQL argument list> that contains one or more <SQL argument>s that immediately contain a <value expression> that contains a <query name> referencing WQN_j.

3) WQE_i shall not contain a <query expression> *TSQ* that contains a <query name> referencing WQN_j, unless *TSQ* is simply contained in a <derived table> that is immediately contained in a <table primary> that is immediately contained in a <table reference> that is immediately contained in a <from clause> that is immediately contained in a <table expression> that is immediately contained in a <query specification> that constitutes a <simple table> that constitutes a <query primary> that constitutes a <query term> that is immediately contained in a <query expression body> that is WQE_i.

4) WQE_i shall not contain a <query specification> *QS* such that:

A) *QS* immediately contains a <table expression> *TE* that contains a <query name> referencing WQN_j, and

B) *QS* immediately contains a <select list> *SL* or *TE* immediately contains a <having clause> *HC* and *SL* or *TE* contain a <set function specification>.

5) WQE_i shall not contain a <query expression body> that contains a <query name> referencing WQN_j and simply contains INTERSECT ALL or EXCEPT ALL.

6) WQE_i shall not contain a <qualified join> *QJ* in which:

A) *QJ* immediately contains a <join type> that specifies FULL and a <table reference> or <table factor> that contains a <query name> referencing WQN_j.

B) *QJ* immediately contains a <join type> that specifies LEFT and a <table factor> following the <join type> that contains a <query name> referencing WQN_j.

C) *QJ* immediately contains a <join type> that specifies RIGHT and a <table reference> preceding the <join type> that contains a <query name> referencing WQN_j.

7) WQE_i shall not contain a <natural join> *QJ* in which:

A) *QJ* immediately contains a <join type> that specifies FULL and a <table reference> or <table primary> that contains a <query name> referencing WQN_j.

B) *QJ* immediately contains a <join type> that specifies LEFT and a <table primary> following the <join type> that contains a <query name> referencing WQN_j.

C) *QJ* immediately contains a <join type> that specifies RIGHT and a <table reference> preceding the <join type> that contains a <query name> referencing WQN_j.

iv) If WLE_i is recursive, then WLE_i shall be linearly recursive.

v) WLE_i is said to be *expandable* if all of the following are true:

1) WLE_i is recursive.

2) WLE_i is linearly recursive.

3) WQE_i is a <query expression body> that immediately contains UNION or UNION ALL. Let $WQEB_i$ be the <query expression body> immediately contained in WQE_i. Let QEL_i and QTR_i be the <query expression body> and the <query term> immediately contained in $WQEB_i$. WQN_i shall not be contained in QEL_i, and QTR_i shall be a <query specification>.

4) WQN_i is not mutually recursive.

h) If a <with list element> *WLE* is not expandable, then it shall not immediately contain a <search or cycle clause>.

3) Let *T* be the table specified by the <query expression>.

4) The <explicit table>

```
TABLE <table or query name>
```

is equivalent to the <query expression>

```
( SELECT * FROM <table or query name> )
```

5) Let *set operator* be UNION ALL, UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT.

6) If UNION, EXCEPT, or INTERSECT is specified and neither ALL nor DISTINCT is specified, then DISTINCT is implicit.

7) <query expression> *QE1* is *simply updatable* if, for every <query expression> or <query specification> *QE2* that is simply contained in the <query expression body> of *QE1*, all of the following are true:

a) *QE1* contains *QE2* without an intervening <query expression body> that specifies UNION ALL, UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.

b) *QE1* contains *QE2* without an intervening <query term> that specifies INTERSECT.

c) *QE2* is simply updatable.

8) <query expression> *QE1* is *updatable* if for every <simple table> *QE2* that is simply contained in *QE1*:

a) *QE2* is not a <table value constructor>.

b) *QE1* contains *QE2* without an intervening <query expression body> that specifies UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.

c) If *QE1* simply contains a <query expression body> *QEB* that specifies UNION ALL, then:

i) *QEB* immediately contains a <query expression body> *LO* and a <query term> *RO* such that no leaf generally underlying table of *LO* is also a leaf generally underlying table of *RO*.

ii) For every column of *QEB*, the underlying columns in the tables identified by *LO* and *RO*, respectively, are either both updatable or not updatable.

d) *QE1* contains *QE2* without an intervening <query term> that specifies INTERSECT.

e) *QE2* is updatable.

9) A table specified by a <query name> immediately contained in a <with list element> *WLE* is *updatable* if and only if the <query expression> simply contained in *WLE* is updatable.

10) A table specified by a <query name> immediately contained in a <with list element> *WLE* is *simply updatable* if and only if the <query expression> simply contained in *WLE* is simply updatable.

11) <query expression> *QE1* is *insertable-into* if the <query expression body> of *QE1* is a <query primary> that is one of the following:

a) An insertable-into <query specification>.

b) An <explicit table> that identifies a table that is insertable-into.

c) Of the form <left paren> <query expression body> <right paren>, where the parenthesized <query expression body> recursively satisfies this condition.

12) A table specified by a <query name> immediately contained in a <with list element> *WLE* is *insertable-into* if the <query expression> simply contained in *WLE* is insertable-into.

13) For every <simple table> *ST* contained in *QE*,

Case:

a) If *ST* is a <query specification> *QS*, then the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of *QS*.

b) If *ST* is an <explicit table> *ET*, then the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of the table identified by the <table or query name> contained in *ET*.

c) Otherwise, the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of the <table value constructor> immediately contained in *ST*.

14) For every <query primary> *QP* contained in *QE*,

Case:

a) If *QP* is a <simple table> *ST*, then the column descriptor of each column of *QP* is the same as the column descriptor of the corresponding column of *ST*.

b) Otherwise, the column descriptor of each column of *QP* is the same as the column descriptor of the corresponding column of the <query expression body> immediately contained in *QP*.

15) If a set operator is specified in a <query term> or a <query expression body>, then:

a) Let *T1*, *T2*, and *TR* be respectively the first operand, the second operand, and the result of the <query term> or <query expression body>.

b) Let *TN1* and *TN2* be the effective names for *T1* and *T2*, respectively.

c) If the set operator is UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT, then each column of *T1* and *T2* is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

16) If a set operator is specified in a <query term> or a <query expression body>, then let *OP* be the set operator.

Case:

a) If CORRESPONDING is specified, then:

i) Within the columns of *T1*, equivalent <column name>s shall not be specified more than once and within the columns of *T2*, equivalent <column name>s shall not be specified more than once.

ii) At least one column of *T1* shall have a <column name> that is the <column name> of some column of *T2*.

iii) Case:

1) If <corresponding column list> is not specified, then let *SL* be a <select list> of those <column name>s that are <column name>s of both *T1* and *T2* in the order that those <column name>s appear in *T1*.

2) If <corresponding column list> is specified, then let *SL* be a <select list> of those <column name>s explicitly appearing in the <corresponding column list> in the order that these <column name>s appear in the <corresponding column list>. Every <column name> in the <corresponding column list> shall be a <column name> of both *T1* and *T2*.

iv) The <query term> or <query expression body> is equivalent to:

```
( SELECT SL FROM TN1 ) OP ( SELECT SL FROM TN2 )
```

b) If CORRESPONDING is not specified, then *T1* and *T2* shall be of the same degree.

17) If a <query term> is a <query primary>, then the declared type of the <query term> is that of the <query primary>. The column descriptor of the *i*-th column of the <query term> is the same as the column descriptor of the *i*-th column of the <query primary>.

18) If a <query term> immediately contains a set operator, then:

a) Let *C* be the <column name> of the *i*-th column of *T1*. If the <column name> of the *i*-th column of *T2* is *C*, then the <column name> of the *i*-th column of *TR* is *C*; otherwise, the <column name> of the *i*-th column of *TR* is implementation-dependent.

b) The declared type of the *i*-th column of *TR* is determined by applying Subclause 9.3, "Result of data type combinations", to the declared types of the *i*-th column of *T1* and the *i*-th column of *T2*. If the *i*-th columns of either *T1* or *T2* are known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

19) If a <query term> is a <query primary>, then the column descriptors of the <query term> are the same as the column descriptors of the <query primary>.

20) Case:

a) If a <query expression body> is a <query term>, then the column descriptors of the <query expression body> are the same as the column descriptors of the <query term>.

b) If a <query expression body> immediately contains a set operator, then:

i) Let *C* be the <column name> of the *i*-th column of *T1*. If the <column name> of the *i*-th column of *T2* is *C*, then the <column name> of the *i*-th column of *TR* is *C*; otherwise, the <column name> of the *i*-th column of *TR* is implementation-dependent.

ii) If *TR* is not the result of an anchor expression, then the declared type of the *i*-th column of *TR* is determined by applying the Syntax Rules of Subclause 9.3, "Result of data type combinations", to the declared types of the *i*-th column of *T1* and the *i*-th column of *T2*.

Case:

1) If the <query expression body> immediately contains EXCEPT, then if the *i*-th column of *T1* is known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

2) Otherwise, if the *i*-th columns of both *T1* and *T2* are known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

iii) If *TR* is the result of an anchor expression *ARE*, then:

1) Let *l* be the number of recursive tables that belong to the stratum of *ARE*. For *j* ranging from 1 (one) to *l*, let WQT_j be those tables. Of the operands *T1* and *T2* of *TR*, let *TNREC* be the operand that is the result of the non-recursive operand of *ARE* and let *TREC* be the other operand. The *i*-th column of *TR* is said to be *recursively referred to* if there exists at least one *k*, $1 \le k \le l$, such that a column of WQT_k is an underlying column of the *i*-th column of *TREC*. Otherwise, that column is said to be *not recursively referred to*.

2) If the *i*-th column of *TR* is not recursively referred to, then the declared type of the *i*-th column of *TR* is determined by applying Subclause 9.3, "Result of data type combinations", to the declared types of the *i*-th column of *T1* and the *i*-th column of *T2*. If the *i*-th columns of either *T1* or *T2* are known not nullable, then the *i*-th column of *TR* is *known not nullable*; otherwise, the *i*-th column of *TR* is *possibly nullable*.

3) If the *i*-th column of *TR* is recursively referred to, then:

A) The *i*-th column of *TR* is *possibly nullable*.

B) Case:

I) If *T1* is *TNREC*, then if the *i*-th column of *TR* is recursively referred to, then the declared type of the *i*-th column of *TR* is the same as the declared type of the *i*-th column of *T1*.

II) If *T2* is *TNREC*, then if the *i*-th column of *TR* is recursively referred to, then the declared type of the *i*-th column of *TR* is the same as the declared type of the *i*-th column of *T2*.

21) The *simply underlying table*s of *QE* are the <table or query name>s, <query specification>s, and <derived table>s contained, without an intervening <derived table> or an intervening <join condition>, in the <query expression body> immediately contained in *QE*.

22) An <explicit table> is *possibly non-deterministic* if the simply contained <table or query name> identifies a viewed table whose original <query expression> is possibly non-deterministic.

23) A <query expression> is *possibly non-deterministic* if any of the following are true:

a) The <query expression> is a <query primary> that is possibly non-deterministic.

b) UNION, EXCEPT, or INTERSECT is specified and either of the first or second operands is possibly non-deterministic.

c) UNION, EXCEPT, or INTERSECT is specified and there is a column of the result such that the declared types *DT1* and *DT2* of the column in the two operands have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

d) Both of the following are true:

   i) *T* contains a set operator UNION and ALL is not specified, or *T* contains either of the set operators EXCEPT or INTERSECT.

   ii) Exactly one of the following is true:

      1) The first or second operand contains a column that has a declared type of character string.

      2) The first or second operand contains a column that has a declared type of datetime with time zone.

      3) The first or second operand contains a column that has a declared type that is a user-defined type.

24) The *underlying columns* of each column of *QE* and of *QE* itself are defined as follows:

a) A column of a <table value constructor> has no underlying columns.

b) The underlying columns of every *i*-th column of a <simple table> *ST* are the underlying columns of the *i*-th column of the table immediately contained in *ST*.

c) If no set operator is specified, then the underlying columns of every *i*-th column of *QE* are the underlying columns of the *i*-th column of the <simple table> simply contained in *QE*.

d) If a set operator is specified, then the underlying columns of every *i*-th column of *QE* are the underlying columns of the *i*-th column of *T1* and those of the *i*-th column of *T2*.

e) Let *C* be some column. *C* is an underlying column of *QE* if and only if *C* is an underlying column of some column of *QE*.

25) The *updatable columns* of *QE* are defined as follows:

a) A column of a <table value constructor> is not an updatable column.

b) A column of a <simple table> is an *updatable column* of *ST* if the underlying column of *ST* is updatable.

c) If no set operator is specified, then a column of *QE* is an *updatable column* of *QE* if its underlying column is updatable.

d) If a set operator is specified, then

   Case:

   i) If the SQL implementation supports Feature T111, “Updatable joins, unions, and columns”, a set operator UNION ALL is specified, and both underlying columns of the *i*-th column of *QE* are updatable, then the *i*-th column of *QE* is an updatable column of *QE*.

   ii) Otherwise, the *i*-th column of *QE* is not updatable.

NOTE 192 — If a set operator UNION DISTINCT, EXCEPT, or INTERSECT is specified, or if the SQL implementation does not support Feature T111, "Updatable joins, unions and columns", then there are no updatable columns.

26) A <query expression> *QE* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

Access Rules

None.

General Rules

1) If a non-recursive <with clause> is specified, then:

   a) For every <with list element> *WLE*, let *WQN* be the <query name> immediately contained in *WLE*. Let *WQE* be the <query expression> immediately contained in *WLE*. Let *WLT* be the table resulting from evaluation of *WQE*, with each column name replaced by the corresponding element of the <with column list>, if any, immediately contained in *WLE*.

   b) Every <table reference> contained in <query expression> that specifies *WQN* identifies *WLT*.

2) If a potentially recursive <with clause> *WC* is specified, then:

   a) Let *n* be the number of <with list element>s WLE_i of the <with list> *WL* immediately contained in *WC*. For *i* ranging from 1 (one) to *n*, let WQN_i and WQE_i be the <query name>s and the <query expression>s immediately contained in WLE_i. Let WLP_j be the elements of a partitioning of *WL* such that each WLP_j contains all WLE_i that belong to one stratum, and let *m* be the number of partitions. Let the *partition dependency graph PDG* of *WL* be a directed graph such that:

      i) Each partition WLP_j of *WL* is represented by exactly one node of *PDG*.

      ii) There is an arc from the node representing WLP_j to the node representing WLP_k if and only if WLP_j contains at least one WLE_i, WLP_k contains at least one WLE_h, and WQE_i contains a <query name> referencing WQN_h.

   b) While the set of nodes of *PDG* is not empty, do:

      i) Evaluate the partitions of *PDG* that have no outgoing arc.

      ii) Remove the partitions and their incoming arcs from *PDG*.

   c) Let *LIP* be some partition of *WL*. Let *m* be the number of <with list element>s in *LIP*, and for *i* ranging from 1 (one) to *m*, let WLE_i be a <with list element> of *LIP*, and let WQN_i and WQE_i be the <query name> and <query expression> immediately contained in WLE_i. Let SQE_i be the set of <query expression>s contained in WQE_i. Let *SQE* be a set of <query expression>s such that a <query expression> belongs to *SQE* if and only if it is contained in some WQE_i. Let *p* be the number of <query expression>s in *SQE* and let AQE_i, $1 \le k \le p$ be the *k*-th <query expression> belonging to *SQE*.

      i) Every <query expression> AQE_k that contains a recursive query name in scope is marked as *recursive*.

ii) Let RT_k and WT_k be tables whose row type is the row type of AQE_k. Let RT_k and WT_k be initially empty. RT_k and WT_k are said to be *associated with* AQE_k. If AQE_k is immediately contained in some WQE_i, then RT_k and WT_k are said to be the *intermediate result table* and *working table*, respectively, *associated with* the <query name> WQN_i.

iii) If a <query expression> AQE_k not marked as recursive is immediately contained in a <query expression body> that is marked as recursive and that specifies UNION, then AQE_i is marked as *iteration ignorable*.

iv) For each AQE_k,

Case:

1) If AQE_k consists of a <query specification> that immediately contains DISTINCT, then AQE_k *suppresses duplicates*.

2) If AQE_k consists of a <query expression body> or <query term> that explicitly or implicitly immediately contains DISTINCT, then AQE_k *suppresses duplicates*.

3) Otherwise, AQE_k does not suppress duplicates.

v) If an AQE_k is not marked as recursive, then let RT_k and WT_k be the result of AQE_k.

vi) For every RT_k, let RTN_k be the name of RT_k. If AQE_k is not marked as recursive, then replace AQE_k with:

```
TABLE RTN_k
```

vii) For every WQE_i of *LIP*, let the *recursive query names in scope* denote the associated result tables. Evaluate every WQE_i. For every AQE_k contained in any such WQE_i, let RT_k and WT_k be the result of AQE_k.

NOTE 193 — This ends the initialization phase of the evaluation of a partition.

viii) For every AQE_k of *LIP* that is marked as iteration ignorable, let RT_k be an empty table.

ix) While some WT_k of *LIP* is not empty, do:

1) Let the recursive query names in scope of *LIP* denote the associated working tables.

2) Evaluate every WQE_i of *LIP*.

3) For every AQE_k that is marked as recursive,

Case:

A) If AQE_k suppresses duplicates, then let WT_k be the result of AQE_k EXCEPT RTN_k.

B) Otherwise, let WT_k be the result of AQE_k.

4) For every WT_k, let WTN_k be the table name of WT_k. Let RT_k be the result of:

```
TABLE WTN_k UNION ALL TABLE RTN_k
```

x) Any reference to WQN_i identifies the intermediate result table RT_k associated with WQN_i.

3) Case:

a) If no set operator is specified, then *T* is the result of the specified <simple table>.

b) Otherwise:

i) Let *D* be the degree of *T*.

ii) For each *i*, 1 (one) ≤ *i* ≤ *D*:

1) Let DTC_i be the declared type of the *i*-th column of *T*.

2) Let $TCN1_i$ be the effective name for the *i*-th column of *T1*.

3) Let $TCN2_i$ be the effective name for the *i*-th column of *T2*.

4) Let *ET1* be a <query expression> of the form

```
SELECT CAST(TCN1_1 AS DTC_1),
       CAST TCN1_1 AS DTC_1),
       ...,
       CAST TCN1_D AS DTC_D)
FROM TN1
```

5) Let *ET2* be a <query expression> of the form

```
SELECT CAST(TCN2_1 AS DTC_1),
       CAST TCN2_2 AS DTC_2),
       ...,
       CAST TCN2_D AS DTC_D)
FROM TN2
```

iii) *T* contains the following rows:

1) Let *R* be a row that is a duplicate of some row in *ET1* or of some row in *ET2* or both. Let *m* be the number of duplicates of *R* in *ET1* and let *n* be the number of duplicates of *R* in *ET2*, where *m* ≥ 0 and *n* ≥ 0.

2) If DISTINCT is specified or implicit, then

Case:

A) If UNION is specified, then

Case:

I) If *m* > 0 or *n* > 0, then *T* contains exactly one duplicate of *R*.

II) Otherwise, *T* contains no duplicate of *R*.

B) If EXCEPT is specified, then

Case:

I) If *m* > 0 and *n* = 0, then *T* contains exactly one duplicate of *R*.

II) Otherwise, *T* contains no duplicate of *R*.

C) If INTERSECT is specified, then

Case:

I) If $m > 0$ and $n > 0$, then *T* contains exactly one duplicate of *R*.

II) Otherwise, *T* contains no duplicates of *R*.

3) If ALL is specified, then

Case:

A) If UNION is specified, then the number of duplicates of *R* that *T* contains is $(m + n)$.

B) If EXCEPT is specified, then the number of duplicates of *R* that *T* contains is the maximum of $(m - n)$ and 0 (zero).

C) If INTERSECT is specified, then the number of duplicates of *R* that *T* contains is the minimum of *m* and *n*.

NOTE 194 — See the General Rules of Subclause 8.2, "<comparison predicate>".

4) Case:

a) If EXCEPT is specified and a row *R* of *T* is replaced by some row *RR*, then the row of *T1* from which *R* is derived is replaced by *RR*.

b) If INTERSECT is specified, then:

i) If a row *R* is inserted into *T*, then:

1) If *T1* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T1*.

2) If *T1* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T1*.

3) If *T2* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T2*.

4) If *T2* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T2*.

ii) If a row *R* is replaced by some row *RR*, then:

1) The row of *T1* from which *R* is derived is replaced with *RR*.

2) The row of *T2* from which *R* is derived is replaced with *RR*.

Conformance Rules

1) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", in conforming SQL language, a <query expression> shall not contain a <with clause>.

2) Without Feature T122, "WITH (excluding RECURSIVE) in subquery", in conforming SQL language, a <query expression> contained in a <query expression> shall not contain a <with clause>.

3) Without Feature T131, "Recursive query", conforming SQL language shall not contain a <query expression> that contains RECURSIVE.

4) Without Feature T132, “Recursive query in subquery”, in conforming SQL language, a <query expression> contained in a <query expression> shall not contain RECURSIVE.

5) Without Feature F661, “Simple tables”, conforming SQL language shall not contain a <simple table> that immediately contains a <table value constructor> except in an <insert statement>.

6) Without Feature F661, “Simple tables”, conforming SQL language shall not contain an <explicit table>.

7) Without Feature F302, “INTERSECT table operator”, conforming SQL language shall not contain a <query term> that contains INTERSECT.

8) Without Feature F301, “CORRESPONDING in query expressions”, conforming SQL language shall not contain a <query expression> that contains CORRESPONDING.

9) Without Feature T551, “Optional key words for default syntax”, conforming SQL language shall not contain UNION DISTINCT, EXCEPT DISTINCT, or INTERSECT DISTINCT.

10) Without Feature F304, “EXCEPT ALL table operator”, conforming SQL language shall not contain a <query expression> that contains EXCEPT ALL.

NOTE 195 — If DISTINCT, INTERSECT or EXCEPT is specified, then the Conformance Rules of Subclause 9.10, “Grouping operations”, apply.

7.14 <search or cycle clause>

Function

Specify the generation of ordering and cycle detection information in the result of recursive query expressions.

Format

```
<search or cycle clause> ::=
    <search clause>
  | <cycle clause>
  | <search clause> <cycle clause>

<search clause> ::=
  SEARCH <recursive search order> SET <sequence column>

<recursive search order> ::=
    DEPTH FIRST BY <column name list>
  | BREADTH FIRST BY <column name list>

<sequence column> ::=
  <column name>

<cycle clause> ::=
  CYCLE <cycle column list> SET <cycle mark column> TO <cycle mark value>
      DEFAULT <non-cycle mark value> USING <path column>

<cycle column list> ::=
  <cycle column> [ { <comma> <cycle column> }... ]

<cycle column> ::=
  <column name>

<cycle mark column> ::=
  <column name>

<path column> ::=
  <column name>

<cycle mark value> ::=
  <value expression>

<non-cycle mark value> ::=
  <value expression>
```

Syntax Rules

1) Let *WLEC* be an expandable <with list element> immediately containing a <search or cycle clause>.

2) Let *WQN* be the <query name>, *WCL* the <with column list>, and *WQE* the <query expression> immediately contained in *WLEC*. Let *WQEB* be the <query expression body> immediately contained in *WQE*. Let *OP* be the set operator immediately contained in *WQEB*. Let *TLO* be the <query expression body> that constitutes the first operand of *OP* and let *TRO* be the <query specification> that (necessarily) constitutes the second operand of *OP*.

a) Let *TROSL* be the <select list> immediately contained in *TRO*. Let *WQNTR* be the <table reference> simply contained in the <from clause> immediately contained in the <table expression> *TROTE* immediately contained in *TRO* such that *WQNTR* immediately contains *WQN*.

Case:

i) If *WQNTR* simply contains a <correlation name>, then let *WQNCRN* be that <correlation name>.

ii) Otherwise, let *WQNCRN* be *WQN*.

b) Case:

i) If *WLEC* simply contains a <search clause> *SC*, then let *SQC* be the <sequence column> and *SO* be the <recursive search order> immediately contained in *SC*. Let *CNL* be the <column name list> immediately contained in *SO*.

1) *WCL* shall not contain a <column name> that is equivalent to *SQC*.

2) Every <column name> of *CNL* shall be equivalent to some <column name> contained in *WCL*. No <column name> shall be contained more than once in *CNL*.

3) Case:

A) If *SO* immediately contains DEPTH, then let *SCEX1* be:

```
WQNCRN.SQC
```

let *SCEX2* be:

```
SQC || ARRAY [ROW(CNL)]
```

and let *SCIN* be:

```
ARRAY [ROW(CNL)]
```

B) If *SO* immediately contains BREADTH, then let *SCEX1* be:

```
( SELECT OC.*
  FROM ( VALUES (WQNCRN.SQC) )
  OC(LEVEL, CNL) )
```

let SCEX2 be:

```
ROW(SQC.LEVEL + 1, CNL)
```

and let *SCIN* be:

```
ROW(0, CNL)
```

ii) If *WLEC* simply contains a <cycle clause> *CC*, then let *CCL* be the <cycle column list>, let *CMC* be the <cycle mark column>, let *CMV* be the <cycle mark value>, let *CMD* be the <non-cycle mark value>, and let *CPA* be the <path column> immediately contained in *CC*.

1) Every <column name> of *CCL* shall be equivalent to some <column name> contained in *WCL*. No <column name> shall be contained more than once in *CCL*.

2) *CMC* and *CPA* shall not be equivalent to each other and not equivalent to any <column name> of *WCL*.

3) The declared type of *CMV* and *CMD* shall be character string of length 1 (one). *CMV* and *CMD* shall be literals and *CMV* shall not be equal to *CMD*.

4) Let *CCEX1* be:

```
WQNCRN.CMC, WQNCRN.CPA
```

Let *CCEX2* be:

```
CASE WHEN ROW(CCL) IN
  (SELECT P.* FROM TABLE(CPA) P)

    THEN CMV ELSE CMD END,

CPA || ARRAY [ROW(CCL)]
```

Let *CCIN* be:

```
CMD, ARRAY [ROW(CCL)]
```

Let *NCCON1* be:

```
CMC <> CMV
```

iii) Case:

1) If *WLEC* simply contains a <search clause> and does not simply contain a <cycle clause>, then let *EWCL* be:

```
WCL, SQC
```

Let *ETLOSL* be:

```
WCL, SCIN
```

Let *ETROSL* be:

```
WCL, SCEX2
```

Let *ETROSL1* be:

```
TROSL, SCEX1
```

Let *NCCON* be:

```
TRUE
```

2) If *WLEC* simply contains a <cycle clause> and does not simply contain a <search clause>, then let *EWCL* be:

```
WCL, CMC, CPA
```

Let *ETLOSL* be:

```
WCL, CCIN
```

Let *ETROSL* be:

```
WCL, CCEX2
```

Let *ETROSL1* be:

```
TROSL, CCEX1
```

Let *NCCON* be:

```
NCCON1
```

3) If *WLEC* simply contains both a <search clause> and a <cycle clause> *CC*, then:

A) The <column name>s *SQC*, *CMC*, and *CPA* shall not be equivalent to each other.

B) Let *EWCL* be:

```
WCL, SQC, CMC, CPA
```

Let *ETLOSL* be:

```
WCL, SCIN, CCIN
```

Let *ETROSL* be:

```
WCL, SCEX2, CCEX2
```

Let *ETROSL1* be:

```
TROSL, SCEX1, CCEX1
```

C) Let *NCCON* be:

```
NCCON1
```

c) *WLEC* is equivalent to the expanded <with list element>:

```
WQN(EWCL) AS
  ( SELECT ETLOSL FROM (TLO) TLOCRN(WCL)
    OP
    SELECT ETROSL
    FROM (SELECT ETROSL1 TROTE) TROCRN(EWCL)
    WHERE NCCON
  )
```

Access Rules

None.

General Rules

None.

Conformance Rules

None.

7.15 <subquery>

Function

Specify a scalar value, a row, or a table derived from a <query expression>.

Format

```
<scalar subquery> ::=
  <subquery>

<row subquery> ::=
  <subquery>

<table subquery> ::=
  <subquery>

<subquery> ::=
  <left paren> <query expression> <right paren>
```

Syntax Rules

1) The degree of a <scalar subquery> shall be 1 (one).

2) The degree of a <row subquery> shall be greater than 1 (one).

3) Let *QE* be the <query expression> simply contained in <subquery>.

4) The declared type of a <scalar subquery> is the declared type of the column of *QE*.

5) The declared type of a <row subquery> is a row type consisting of one field for each column of *QE*. The declared type and field name of each field of this row type is the declared type and column name of the corresponding column of *QE*.

6) The declared types of the columns of a <table subquery> are the declared types of the respective columns of *QE*.

Access Rules

None.

General Rules

1) Let *OLDSEC* be the most recent statement execution context. A new statement execution context *NEWSEC* is established. *NEWSEC* becomes the most recent statement execution context and is atomic.

2) Let *RS* be a <row subquery>. Let *RRS* be the result of the <query expression> simply contained in *RS*. Let *D* be the degree of *RRS*.

   Case:

a) If the cardinality of *RRS* is greater than 1 (one), then an exception condition is raised: *cardinality violation*.

b) If the cardinality of *RRS* is 0 (zero), then the value of the <row subquery> is a row whose degree is *D* and whose fields are all the null value.

c) Otherwise, the value of *RS* is *RRS*.

3) Let *SS* be a <scalar subquery>.

Case:

a) If the cardinality of *SS* is greater than 1 (one), then an exception condition is raised: *cardinality violation*.

b) If the cardinality of *SS* is 0 (zero), then the value of the <scalar subquery> is the null value.

c) Otherwise, let *C* be the column of <query expression> simply contained in *SS*. The value of *SS* is the value of *C* in the unique row of the result of the <scalar subquery>.

4) All savepoints that were established during the existence of *NEWSEC* are destroyed. *NEWSEC* ceases to exist and *OLDSEC* becomes the most recent statement execution context.

Conformance Rules

None.

(Blank page)

8 Predicates

This Clause is modified by Clause 7, "Predicates", in ISO/IEC 9075-13.
This Clause is modified by Clause 8, "Predicates", in ISO/IEC 9075-14.

8.1 <predicate>

This Subclause is modified by Subclause 8.1, "<predicate>", in ISO/IEC 9075-14.

Function

Specify a condition that can be evaluated to give a boolean value.

Format

```
<predicate> ::=
    <comparison predicate>
  | <between predicate>
  | <in predicate>
  | <like predicate>
  | <similar predicate>
  | <regex like predicate>
  | <null predicate>
  | <quantified comparison predicate>
  | <exists predicate>
  | <unique predicate>
  | <normalized predicate>
  | <match predicate>
  | <overlaps predicate>
  | <distinct predicate>
  | <member predicate>
  | <submultiset predicate>
  | <set predicate>
  | <type predicate>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) The result of a <predicate> is the truth value of the immediately contained <comparison predicate>, <between predicate>, <in predicate>, <like predicate>, <similar predicate>, <regex like predicate>, <null predicate>, <quantified comparison predicate>, <exists predicate>, <unique predicate>, <match predicate>, <overlaps predicate>, <distinct predicate>, <member predicate>, <submultiset predicate>, <set predicate>, or <type predicate>.

Conformance Rules

None.

8.2 <comparison predicate>

This Subclause is modified by Subclause 7.1, "<comparison predicate>", in ISO/IEC 9075-13.

Function

Specify a comparison of two row values.

Format

```
<comparison predicate> ::=
  <row value predicand> <comparison predicate part 2>

<comparison predicate part 2> ::=
  <comp op> <row value predicand>

<comp op> ::=
    <equals operator>
  | <not equals operator>
  | <less than operator>
  | <greater than operator>
  | <less than or equals operator>
  | <greater than or equals operator>
```

Syntax Rules

1) The two <row value predicand>s shall be of the same degree.

2) Let *corresponding fields* be fields with the same ordinal position in the two <row value predicand>s.

3) The declared types of the corresponding fields of the two <row value predicand>s shall be comparable.

4) Let R_x and R_y respectively denote the first and second <row value predicand>s.

5) Let N be the number of fields in the declared type of R_x. Let X_i, 1 (one) ≤ i ≤ N, be the i-th field in the declared type of R_x and let Y_i be the i-th field in the declared type of R_y. For each i:

   a) Case:

      i) If <comp op> is <equals operator> or <not equals operator>, then X_i and Y_i are operands of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

      ii) Otherwise, X_i and Y_i are operands of an ordering operation. The Syntax Rules of Subclause 9.12, "Ordering operations", apply.

   b) Case:

      i) If the declared types of X_i and Y_i are user-defined types, then let *UDT1* and *UDT2* be respectively the declared types of X_i and Y_i. *UDT1* and *UDT2* shall be in the same subtype family. *UDT1* and *UDT2* shall have comparison types.

      NOTE 196 — "Comparison type" is defined in Subclause 4.7.6, "User-defined type comparison and assignment".

NOTE 197 — The comparison form and comparison categories included in the user-defined type descriptors of both *UDT1* and *UDT2* are constrained to be the same and to be the same as those of all their supertypes. If the comparison category is either STATE or RELATIVE, then *UDT1* and *UDT2* are constrained to have the same comparison function; if the comparison category is MAP, they are not constrained to have the same comparison function.

ii) If the declared types of X_i and Y_i are reference types, then the referenced type of the declared type of X_i and the referenced type of the declared type of Y_i shall have a common supertype.

iii) If the declared types of X_i and Y_i are collection types in which the declared type of the elements are ET_x and ET_y, respectively, then let *RV1* and *RV2* be <value expression>s whose declared types are respectively ET_x and ET_y. The Syntax Rules of this Subclause are applied to:

```
RV1 <comp op> RV2
```

iv) If the declared types of X_i and Y_i are row types, then let *RV1* and *RV2* be <value expression>s whose declared types are respectively that of X_i and Y_i. The Syntax Rules of this Subclause are applied to:

```
RV1 <comp op> RV2
```

6) Let *CP* be the <comparison predicate> "R_x <comp op> R_y".

Case:

a) If the <comp op> is <not equals operator>, then *CP* is equivalent to:

```
NOT(Rx = Ry )
```

b) If the <comp op> is <greater than operator>, then *CP* is equivalent to:

```
(Ry < Rx )
```

c) If the <comp op> is <less than or equals operator>, then *CP* is equivalent to:

```
(Rx < Ry
    OR
Ry = Rx )
```

d) If the <comp op> is <greater than or equals operator>, then *CP* is equivalent to:

```
(Ry < Rx
    OR
Ry = Rx )
```

Access Rules

None.

General Rules

1) Let *XV* and *YV* be two values represented by <value expression>s *X* and *Y*, respectively. The result of:

```
X <comp op> Y
```

is determined as follows.

Case:

a) If either *XV* or *YV* is the null value, then

```
X <comp op> Y
```

is *Unknown*.

b) Otherwise,

Case:

i) If the declared types of *XV* and *YV* are row types with degree *N*, then let X_i, 1 (one) ≤ i ≤ N, denote a <value expression> whose value and declared type is that of the *i*-th field of *XV* and let Y_i denote a <value expression> whose value and declared type is that of the *i*-th field of *YV*. The result of

```
X <comp op> Y
```

is determined as follows:

1) $X = Y$ is *True* if and only if $X_i = Y_i$ is *True* for all i.

2) $X < Y$ is *True* if and only if $X_i = Y_i$ is *True* for all $i < n$ and $X_n < Y_n$ for some n.

3) $X = Y$ is *False* if and only if NOT ($X_i = Y_i$) is *True* for some i.

4) $X < Y$ is *False* if and only if $X = Y$ is *True* or $Y < X$ is *True*.

5) X <comp op> Y is *Unknown* if X <comp op> Y is neither *True* nor *False*.

ii) If the declared types of *XV* and *YV* are array types and the cardinalities of *XV* and *YV* are *N1* and *N2*, respectively, then let X_i, 1 (one) ≤ i ≤ *N1*, denote a <value expression> whose value and declared type is that of the *i*-th element of *XV* and let Y_i denote a <value expression> whose value and declared type is that of the *i*-th element of *YV*. The result of

```
X <comp op> Y
```

is determined as follows:

1) $X = Y$ is *True* if *N1* = 0 (zero) and *N2* = 0 (zero).

2) $X = Y$ is *True* if *N1* = *N2* and, for all i, $X_i = Y_i$ is *True*.

3) $X = Y$ is *False* if and only if *N1* ≠ *N2* or NOT ($X_i = Y_i$) is *True*, for some i.

4) X <comp op> Y is *Unknown* if X <comp op> Y is neither *True* nor *False*.

iii) If the declared types of *XV* and *YV* are multiset types and the cardinalities of *XV* and *YV* are *N1* and *N2*, respectively, then the result of

```
X <comp op> Y
```

is determined as follows.

Case:

1) $X = Y$ is <u>*True*</u> if $N1 = 0$ (zero) and $N2 = 0$ (zero).

2) $X = Y$ is <u>*True*</u> if $N1 = N2$, and there exist an enumeration XVE_j, 1 (one) $\le j \le N1$, of the elements of XV and an enumeration YVE_j, 1 (one) $\le j \le N1$, of the elements of YV such that for all j, $XVE_j = YVE_j$.

3) $X = Y$ is <u>*Unknown*</u> if $N1 = N2$, and there exist an enumeration XVE_j, 1 (one) $\le j \le N1$, of the elements of XV and an enumeration YVE_j, 1 (one) $\le j \le N1$, of the elements of YV such that for all j, "$XVE_j = YVE_j$" is either <u>*True*</u> or <u>*Unknown*</u>.

4) Otherwise, $X = Y$ is <u>*False*</u>.

iv) If the declared types of XV and YV are user-defined types, then let UDT_x and UDT_y be respectively the declared types of XV and YV. The result of

```
X <comp op> Y
```

is determined as follows:

1) If the comparison category of UDT_x is MAP, then let *HF1* be the <routine name> with explicit <schema name> of the comparison function of UDT_x and let *HF2* be the <routine name> with explicit <schema name> of the comparison function of UDT_y. If *HF1* identifies an SQL-invoked method, then let *HFX* be *X.HF1*; otherwise, let *HFX* be *HF1*(*X*). If *HF2* identifies an SQL-invoked method, then let *HFY* be *Y.HF2*; otherwise, let *HFY* be *HF2*(*Y*).

```
X <comp op> Y
```

has the same result as

```
HFX <comp op> HFY
```

2) If the comparison category of UDT_x is RELATIVE, then:

A) Let *RF* be the <routine name> with explicit <schema name> of the comparison function of UDT_x.

B) `X = Y`

has the same result as

```
RF (X, Y ) = 0
```

C) `X < Y`

has the same result as

```
RF (X, Y ) = -1
```

D) `X <> Y`

has the same result as

`RF (X, Y ) <> 0`

E) `X > Y`

has the same result as

`RF (X, Y ) = 1`

F) `X <= Y`

has the same result as

`RF (X, Y ) = -1 OR RF (X, Y ) = 0`

G) `X >= Y`

has the same result as

`RF (X, Y ) = 1 OR RF (X, Y ) = 0`

3) If the comparison category of UDT_x is STATE, then:

A) Let *SF* be the <routine name> of the comparison function of UDT_x.

B) `X = Y`

has the same result as

`SF (X, Y ) = TRUE`

C) `X <> Y`

has the same result as

`SF (X, Y ) = FALSE`

NOTE 198 — Rules for the comparison of user-defined types in which <comp op> is other than <equals operator> or <less than operator> are included for informational purposes only, since such predicates are equivalent to other <comparison predicate>s whose <comp op> is <equals operator> or <less than operator>.

v) Otherwise, the result of

`X <comp op> Y`

is *True* or *False* as follows:

1) `X = Y`

is *True* if and only if *XV* and *YV* are equal.

2) `X < Y`

is *True* if and only if *XV* is less than *YV*.

3) *X* `<comp op>` *Y*

is *False* if and only if

X `<comp op>` *Y*

is not *True*

2) Numbers are compared with respect to their algebraic value.

3) The comparison of two character strings is determined as follows:

   a) Let *CS* be the collation as determined by Subclause 9.13, "Collation determination", for the declared types of the two character strings.

   b) If the length in characters of *X* is not equal to the length in characters of *Y*, then the shorter string is effectively replaced, for the purposes of comparison, with a copy of itself that has been extended to the length of the longer string by concatenation on the right of one or more pad characters, where the pad character is chosen based on *CS*. If *CS* has the NO PAD characteristic, then the pad character is an implementation-dependent character different from any character in the character set of *X* and *Y* that collates less than any string under *CS*. Otherwise, the pad character is a <space>.

   c) The result of the comparison of *X* and *Y* is given by the collation *CS*.

   d) Depending on the collation, two strings may compare as equal even if they are of different lengths or contain different sequences of characters. When any of the operations MAX, MIN, and DISTINCT reference a grouping column, and the UNION, EXCEPT, and INTERSECT operators refer to character strings, the specific value selected by these operations from a set of such equal values is implementation-dependent.

4) The comparison of two binary string values, *X* and *Y*, of which at least one is a binary large object string value, is determined by comparison of their octets with the same ordinal position. Let X_i and Y_i be the values of the *i*-th octets of *X* and *Y*, respectively, and let L_x be the length in octets of *X* and let L_y be the length in octets of *Y*. *X* is equal to *Y* if and only if $L_x = L_y$ and if $X_i = Y_i$ for all *i*.

5) The comparison of two binary string values *X* and *Y*, neither of which is a binary large object string value, is determined as follows:

   a) Let L_X be the length in octets of *X* and let L_Y be the length in octets of *Y*. Let X_i, 1 (one) $\le i \le L_X$, be the value of the *i*-th octet of *X*, and let Y_i, 1 (one) $\le i \le L_Y$, be the value of the *i*-th octet of *Y*.

   b) If $L_X = L_Y$ and $X_i = Y_i$, 1 (one) $\le i \le L_X$, then *X* is equal to *Y*.

   c) If $L_X < L_Y$, $X_i = Y_i$ for all $i \le L_X$, and the right-most $L_Y - L_X$ octets of *Y* are all X'00's, then it is implementation-defined whether *X* is equal to *Y* or whether *X* is less than *Y*.

   d) If $L_X < L_Y$, $X_i = Y_i$ for all $i \le L_X$, and at least one of the right-most $L_Y - L_X$ octets of *Y* is not X'00', then *X* is less than *Y*.

   e) If $X_j < Y_j$, for some *j*, 0 (zero) < *j* minimum(L_X, L_Y), and $X_i = Y_i$ for all $i < j$, then *X* is less than *Y*.

6) The comparison of two datetimes is determined according to the interval resulting from their subtraction. Let *X* and *Y* be the two values to be compared and let *H* be the least significant <primary datetime field> of *X* and *Y*, including fractional seconds precision if the data type is time or timestamp.

a) *X* is equal to *Y* if and only if

```
( X - Y ) INTERVAL H = INTERVAL '0' H
```

is *True*.

b) *X* is less than *Y* if and only if

```
( X - Y ) INTERVAL H < INTERVAL '0' H
```

is *True*.

NOTE 199 — Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.6.2, "Datetimes".

7) The comparison of two intervals is determined by the comparison of their corresponding values after conversion to integers in some common base unit. Let *X* and *Y* be the two intervals to be compared. Let *A* TO *B* be the specified or implied datetime qualifier of *X* and *C* TO *D* be the specified or implied datetime qualifier of *Y*. Let *T* be the least significant <primary datetime field> of *B* and *D* and let *U* be a datetime qualifier of the form *T*(*N*), where *N* is an <interval leading field precision> large enough so that significance is not lost in the CAST operation.

Let *XVE* be the <value expression>

```
CAST ( X AS INTERVAL U )
```

Let *YVE* be the <value expression>

```
CAST ( Y AS INTERVAL U )
```

a) *X* is equal to *Y* if and only if

```
CAST ( XVE AS INTEGER ) = CAST ( YVE AS INTEGER )
```

is *True*.

b) *X* is less than *Y* if and only if

```
CAST ( XVE AS INTEGER ) < CAST ( YVE AS INTEGER )
```

is *True*.

8) In comparisons of boolean values, *True* is greater than *False*

9) The result of comparing two reference values *X* and *Y* is determined by the comparison of their octets with the same ordinal position. Let L_x be the length in octets of *X* and let L_y be the length in octets of *Y*. Let X_i and Y_i, 1 (one) $\leq i \leq L_x$, be the values of the *i*-th octets of *X* and *Y*, respectively. *X* is equal to *Y* if and only if $L_x = L_y$ and, for all *i*, $X_i = Y_i$.

Conformance Rules

None.

NOTE 200 — If <comp op> is <equals operator> or <not equals operator>, then the Conformance Rules of Subclause 9.9, "Equality operations", apply. Otherwise, the Conformance Rules of Subclause 9.12, "Ordering operations", apply.

8.3 <between predicate>

Function

Specify a range comparison.

Format

```
<between predicate> ::=
  <row value predicand> <between predicate part 2>

<between predicate part 2> ::=
  [ NOT ] BETWEEN [ ASYMMETRIC | SYMMETRIC ]
      <row value predicand> AND <row value predicand>
```

Syntax Rules

1) If neither SYMMETRIC nor ASYMMETRIC is specified, then ASYMMETRIC is implicit.

2) Let *X*, *Y*, and *Z* be the first, second, and third <row value predicand>s, respectively.

3) "*X* NOT BETWEEN SYMMETRIC *Y* AND *Z*" is equivalent to "NOT (*X* BETWEEN SYMMETRIC *Y* AND *Z*)".

4) "*X* BETWEEN SYMMETRIC *Y* AND *Z*" is equivalent to "((*X* BETWEEN ASYMMETRIC *Y* AND *Z*) OR (*X* BETWEEN ASYMMETRIC *Z* AND *Y*))".

5) "*X* NOT BETWEEN ASYMMETRIC *Y* AND *Z*" is equivalent to "NOT (*X* BETWEEN ASYMMETRIC *Y* AND *Z*)".

6) "*X* BETWEEN ASYMMETRIC *Y* AND *Z*" is equivalent to "*X*>=*Y* AND *X*<=*Z*".

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T461, "Symmetric BETWEEN predicate", conforming SQL language shall not contain SYMMETRIC or ASYMMETRIC.

NOTE 201 — Since <between predicate> is an ordering operation, the Conformance Rules of Subclause 9.12, "Ordering operations", also apply.

8.4 <in predicate>

Function

Specify a quantified comparison.

Format

```
<in predicate> ::=
  <row value predicand> <in predicate part 2>

<in predicate part 2> ::=
  [ NOT ] IN <in predicate value>

<in predicate value> ::=
    <table subquery>
  | <left paren> <in value list> <right paren>

<in value list> ::=
  <row value expression> [ { <comma> <row value expression> }... ]
```

Syntax Rules

1) If <in value list> consists of a single <row value expression>, then that <row value expression> shall not be a <scalar subquery>.

   NOTE 202 — This Syntax Rule resolves an ambiguity in which <in predicate value> might be interperted either as a <table subquery> or as a <scalar subquery>. The ambiguity is resolved by adopting the interpretation that the <in predicate value> will be interpreted as a <table subquery>.

2) Let *IVL* be an <in value list>.

   (*IVL*)

   is equivalent to the <table value constructor>:

   (VALUES *IVL*)

3) Let *RVC* be the <row value predicand> and let *IPV* be the <in predicate value>.

4) The expression

   RVC NOT IN *IPV*

   is equivalent to

   NOT (*RVC* IN *IPV*)

5) The expression

   RVC IN *IPV*

   is equivalent to

```
RVC = ANY IPV
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F561, “Full value expressions”, conforming SQL language shall not contain a <row value expression> immediately contained in an <in value list> that is not a <value specification>.

NOTE 203 — Since <in predicate> is an equality operation, the Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

8.5 <like predicate>

Function

Specify a pattern-match comparison.

Format

```
<like predicate> ::=
    <character like predicate>
  | <octet like predicate>

<character like predicate> ::=
  <row value predicand> <character like predicate part 2>

<character like predicate part 2> ::=
  [ NOT ] LIKE <character pattern> [ ESCAPE <escape character> ]

<character pattern> ::=
  <character value expression>

<escape character> ::=
  <character value expression>

<octet like predicate> ::=
  <row value predicand> <octet like predicate part 2>

<octet like predicate part 2> ::=
  [ NOT ] LIKE <octet pattern> [ ESCAPE <escape octet> ]

<octet pattern> ::=
  <binary value expression>

<escape octet> ::=
  <binary value expression>
```

Syntax Rules

1) The <row value predicand> immediately contained in <character like predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared types of *CVE*, <character pattern>, and <escape character> shall be character string. *CVE*, <character pattern>, and <escape character> shall be comparable.

2) The <row value predicand> immediately contained in <octet like predicate> shall be a <row value constructor element> that is a <common value expression> *OVE*. The declared types of *OVE*, <octet pattern>, and <escape octet> shall be binary string.

3) If <character like predicate> is specified, then:

   a) Let *MC* be the <character value expression> of *CVE*, let *PC* be the <character value expression> of the <character pattern>, and let *EC* be the <character value expression> of the <escape character> if one is specified.

   b) *MC* NOT LIKE *PC*

is equivalent to

```
NOT (MC LIKE PC)
```

c) *MC* `NOT LIKE` *PC* `ESCAPE` *EC*

is equivalent to

```
NOT (MC LIKE PC ESCAPE EC)
```

d) The collation used for <like predicate> is determined by applying Subclause 9.13, "Collation determination", with operands *CVE*, *PC*, and (if specified) *EC*.

It is implementation-defined which collations can be used as collations for the <like predicate>.

4) If <octet like predicate> is specified, then:

a) Let *MB* be the <binary value expression> of the *OVE*, let *PB* be the <binary value expression> of the <octet pattern>, and let *EB* be the <binary value expression> of the <escape octet> if one is specified.

b) *MB* `NOT LIKE` *PB*

is equivalent to

```
NOT (MB LIKE PB)
```

c) *MB* `NOT LIKE` *PB* `ESCAPE` *EB*

is equivalent to

```
NOT (MB LIKE PB ESCAPE EB)
```

Access Rules

None.

General Rules

1) Let *MCV* be the value of *MC* and let *PCV* be the value of *PC*. If *EC* is specified, then let *ECV* be its value.

2) Let *MBV* be the value of *MB* and let *PBV* be the value of *PB*. If *EB* is specified, then let *EBV* be its value.

3) If <character like predicate> is specified, then:

a) Case:

i) If ESCAPE is not specified and either *MCV* or *PCV* are null values, then the result of

```
MC LIKE PC
```

is *Unknown*.

ii) If ESCAPE is specified and one or more of *MCV*, *PCV* and *ECV* are null values, then the result of

```
MC LIKE PC ESCAPE EC
```

is *Unknown*.

NOTE 204 — If none of *MCV*, *PCV*, and *ECV* (if present) are null values, then the result is either *True* or *False*.

b) Case:

i) If an <escape character> is specified, then:

1) If the length in characters of *ECV* is not equal to 1, then an exception condition is raised: *data exception — invalid escape character*.

2) If there is not a partitioning of the string *PCV* into substrings such that each substring has length 1 (one) or 2, no substring of length 1 (one) is the escape character *ECV*, and each substring of length 2 is the escape character *ECV* followed by either the escape character *ECV*, an <underscore> character, or the <percent> character, then an exception condition is raised: *data exception — invalid escape sequence*.

If there is such a partitioning of *PCV*, then in that partitioning, each substring with length 2 represents a single occurrence of the second character of that substring and is called a *single character specifier*.

Each substring with length 1 (one) that is the <underscore> character represents an *arbitrary character specifier*. Each substring with length 1 (one) that is the <percent> character represents an *arbitrary string specifier*. Each substring with length 1 (one) that is neither the <underscore> character nor the <percent> character represents the character that it contains and is called a *single character specifier*.

ii) If an <escape character> is not specified, then each <underscore> character in *PCV* represents an arbitrary character specifier, each <percent> character in *PCV* represents an arbitrary string specifier, and each character in *PCV* that is neither the <underscore> character nor the <percent> character represents itself and is called a *single character specifier*.

c) Case:

i) If *MCV* and *PCV* are character strings whose lengths are variable and if the lengths of both *MCV* and *PCV* are 0 (zero), then

```
MC LIKE PC
```

is *True*.

ii) The <predicate>

```
MC LIKE PC
```

is *True* if there exists a partitioning of *MCV* into substrings such that:

1) A substring of *MCV* is a sequence of 0 (zero) or more contiguous characters of *MCV* and each character of *MCV* is part of exactly one substring.

2) If the *i*-th substring of *PCV* is an arbitrary character specifier, then the *i*-th substring of *MCV* is any single character.

3) If the *i*-th substring of *PCV* is an arbitrary string specifier, then the *i*-th substring of *MCV* is any sequence of 0 (zero) or more characters.

4) If the *i*-th substring of *PCV* is a single character specifier, then the *i*-th substring of *MCV* contains exactly 1 (one) character that is equal to the character represented by the single character specifier according to the collation of the <like predicate>.

5) The number of substrings of *MCV* is equal to the number of substring specifiers of *PCV*.

iii) Otherwise,

```
MC LIKE PC
```

is *False*.

4) If <octet like predicate> is specified, then:

a) Case:

i) If ESCAPE is not specified and either *MBV* or *PBV* are null values, then the result of

```
MB LIKE PB
```

is *Unknown*.

ii) If ESCAPE is specified and one or more of *MBV*, *PBV* and *EBV* are null values, then the result of

```
MB LIKE PB ESCAPE EB
```

is *Unknown*.

NOTE 205 — If none of *MBV*, *PBV*, and *EBV* (if present) are null values, then the result is either *True* or *False*.

b) <percent> in the context of an <octet like predicate> has the same bit pattern as the encoding of a <percent> in the SQL_TEXT character set.

c) <underscore> in the context of an <octet like predicate> has the same bit pattern as the encoding of an <underscore> in the SQL_TEXT character set.

d) Case:

i) If an <escape octet> is specified, then:

1) If the length in octets of *EBV* is not equal to 1, then an exception condition is raised: *data exception — invalid escape octet*.

2) If there is not a partitioning of the string *PBV* into substrings such that each substring has length 1 (one) or 2, no substring of length 1 (one) is the escape octet *EBV*, and each substring of length 2 is the escape octet *EBV* followed by either the escape octet *EBV*, an <underscore> octet, or the <percent> octet, then an exception condition is raised: *data exception — invalid escape sequence*.

If there is such a partitioning of *PBV*, then in that partitioning, each substring with length 2 represents a single occurrence of the second octet of that substring. Each substring with length 1 (one) that is the <underscore> octet represents an *arbitrary octet specifier*. Each substring with length 1 (one) that is the <percent> octet represents an *arbitrary string specifier*. Each substring with length 1 (one) that is neither the <underscore> octet nor the <percent> octet represents the octet that it contains.

ii) If an <escape octet> is not specified, then each <underscore> octet in *PBV* represents an arbitrary octet specifier, each <percent> octet in *PBV* represents an arbitrary string specifier, and each octet in *PBV* that is neither the <underscore> octet nor the <percent> octet represents itself.

e) The string *PBV* is a sequence of the minimum number of substring specifiers such that each portion of *PBV* is part of exactly one substring specifier. A *substring specifier* is an arbitrary octet specifier, and arbitrary string specifier, or any sequence of octets other than an arbitrary octet specifier or an arbitrary string specifier.

f) Case:

i) If the lengths of both *MBV* and *PBV* are 0 (zero), then

```
MB LIKE PB
```

is *True*.

ii) The <predicate>

```
MB LIKE PB
```

is *True* if there exists a partitioning of *MBV* into substrings such that:

1) A substring of *MBV* is a sequence of 0 (zero) or more contiguous octets of *MBV* and each octet of *MBV* is part of exactly one substring.

2) If the *i*-th substring specifier of *PBV* is an arbitrary octet specifier, the *i*-th substring of *MBV* is any single octet.

3) the *i*-th substring specifier of *PBV* is an arbitrary string specifier, then the *i*-th substring of *MBV* is any sequence of 0 (zero) or more octets.

4) If the *i*-th substring specifier of *PBV* is an neither an arbitrary character specifier not an arbitrary string specifier, then the *i*-th substring of *MBV* has the same length and bit pattern as that of the substring specifier.

5) The number of substrings of *MBV* is equal to the number of substring specifiers of *PBV*.

iii) Otherwise:

```
MB LIKE PB
```

is *False*.

Conformance Rules

1) Without Feature T042, “Extended LOB data type support”, conforming SQL language shall not contain an <octet like predicate>.

2) Without Feature F281, “LIKE enhancements”, or Feature T022, “Advanced BINARY and VARBINARY data type support”, conforming SQL language shall not contain a <common value expression> simply contained in the <row value predicand> immediately contained in <character like predicate> that is not a column reference.

3) Without Feature F281, “LIKE enhancements”, conforming SQL language shall not contain a <character pattern> that is not a <value specification>.

4) Without Feature F281, “LIKE enhancements”, conforming SQL language shall not contain an <escape character> that is not a <value specification>.

5) Without Feature T042, “Extended LOB data type support”, in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type CHARACTER LARGE OBJECT

6) Without Feature F421, “National character”, and Feature T042, “Extended LOB data type support”, in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.

8.6 <similar predicate>

Function

Specify a character string similarity by means of a regular expression.

Format

```
<similar predicate> ::=
  <row value predicand> <similar predicate part 2>

<similar predicate part 2> ::=
  [ NOT ] SIMILAR TO <similar pattern> [ ESCAPE <escape character> ]

<similar pattern> ::=
  <character value expression>

<regular expression> ::=
    <regular term>
  | <regular expression> <vertical bar> <regular term>

<regular term> ::=
    <regular factor>
  | <regular term> <regular factor>

<regular factor> ::=
    <regular primary>
  | <regular primary> <asterisk>
  | <regular primary> <plus sign>
  | <regular primary> <question mark>
  | <regular primary> <repeat factor>

<repeat factor> ::=
  <left brace> <low value> [ <upper limit> ] <right brace>

<upper limit> ::=
  <comma> [ <high value> ]

<low value> ::=
  <unsigned integer>

<high value> ::=
  <unsigned integer>

<regular primary> ::=
    <character specifier>
  | <percent>
  | <regular character set>
  | <left paren> <regular expression> <right paren>

<character specifier> ::=
    <non-escaped character>
  | <escaped character>

<non-escaped character> ::=
  !! See the Syntax Rules
```

```
<escaped character> ::=
  !! See the Syntax Rules

<regular character set> ::=
    <underscore>
  | <left bracket> <character enumeration>... <right bracket>
  | <left bracket> <circumflex> <character enumeration>... <right bracket>
  | <left bracket> <character enumeration include>...
      <circumflex> <character enumeration exclude>... <right bracket>

<character enumeration include> ::=
  <character enumeration>

<character enumeration exclude> ::=
  <character enumeration>

<character enumeration> ::=
    <character specifier>
  | <character specifier> <minus sign> <character specifier>
  | <left bracket> <colon> <regular character set identifier> <colon> <right bracket>

<regular character set identifier> ::=
  <identifier>
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared types of *CVE*, <similar pattern>, and <escape character> shall be character string. *CVE*, <similar pattern>, and <escape character> shall be comparable.

2) Let *CM* be the <character value expression> of *CVE* and let *SP* be the <similar pattern>. If <escape character> *EC* is specified, then

   ```
   CM NOT SIMILAR TO
   SP ESCAPE EC
   ```

   is equivalent to

   ```
   NOT ( CM SIMILAR TO
   SP ESCAPE EC )
   ```

   If <escape character> *EC* is not specified, then

   ```
   CM NOT SIMILAR TO
   SP
   ```

   is equivalent to

   ```
   NOT ( CM SIMILAR TO
   SP )
   ```

3) The value of the <identifier> that is a <regular character set identifier> shall be either ALPHA, UPPER, LOWER, DIGIT, ALNUM, SPACE, or WHITESPACE.

4) The collation used for <similar predicate> is determined by applying Subclause 9.13, "Collation determination", with operands *CVE*, *PC*, and (if specified) *EC*.

It is implementation-defined which collations can be used as collations for <similar predicate>.

5) A <non-escaped character> is any single character from the character set of the <similar pattern> that is not a <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>, or the character specified by the result of the <character value expression> of <escape character>. A <character specifier> that is a <non-escaped character> represents itself.

6) An <escaped character> is a sequence of two characters: the character specified by the result of the <character value expression> of <escape character>, followed by a second character that is a <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>, or the character specified by the result of the <character value expression> of <escape character>. A <character specifier> that is an <escaped character> represents its second character.

7) The value of <low value> shall be a positive integer. The value of <high value> shall be greater than or equal to the value of <low value>.

Access Rules

None.

General Rules

1) Let *MCV* be the result of the <character value expression> of *CVE* and let *PCV* be the result of the <character value expression> of the <similar pattern>. If *EC* is specified, then let *ECV* be its value.

2) If the result of the <character value expression> of the <similar pattern> is not a zero-length string and does not have the format of a <regular expression>, then an exception condition is raised: *data exception — invalid regular expression*.

3) If an <escape character> is specified, then:

   a) If the length in characters of *ECV* is not equal to 1 (one), then an exception condition is raised: *data exception — invalid escape character*.

   b) If *ECV* is one of <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, or <left brace> and *ECV* occurs in the <regular expression> except in an <escaped character>, then an exception condition is raised: *data exception — invalid use of escape character*.

   c) If *ECV* is a <colon> and the <regular expression> contains a <regular character set identifier>, then an exception condition is raised: *data exception — escape character conflict*.

4) Case:

   a) If ESCAPE is not specified, then if either or both of *MCV* and *PCV* are the null value, then the result of

      ```
      CM SIMILAR TO SP
      ```

      is *Unknown*.

b) If ESCAPE is specified, then if one or more of *MCV*, *PCV*, and *ECV* are the null value, then the result of

```
CM SIMILAR TO SP ESCAPE EC
```

is *Unknown*.

NOTE 206 — If none of *MCV*, *PCV*, and *ECV* (if present) are the null value, then the result is either *True* or *False*.

5) The set of characters in a <character enumeration> is defined as

a) If the enumeration is specified in the form "<character specifier> <minus sign> <character specifier>", then the set of all characters that collate greater than or equal to the character represented by the left <character specifier> and less than or equal to the character represented by the right <character specifier>, according to the collation of the pattern *PCV*.

b) Otherwise, the character that the <character specifier> in the <character enumeration> represents.

6) Let *LV* be the value of the <low value> contained in a <repeat factor> *RF*.

Case:

a) If *RF* does not contain an <upper limit>, then let *HV* be *LV*.

b) If *RF* contains an <upper limit> that contains a <high value>, then let *HV* be the value of <high value>.

c) Otherwise, let *HV* be the length or maximum length of *CVE*.

7) Let *R* be the result of the <character value expression> of the <similar pattern>. The regular language L(*R*) of the <similar pattern> is a (possibly infinite) set of strings. It is defined recursively for well-formed <regular expression>s *Q*, *Q1*, and *Q2* by the following rules:

a) L(*Q1* <vertical bar> *Q2*)

is the union of L(*Q1*) and L(*Q2*)

b) L(*Q* <asterisk>)

is the set of all strings that can be constructed by concatenating zero or more strings from L(*Q*).

c) L(*Q* <plus sign>)

is the set of all strings that can be constructed by concatenating one or more strings from L(*Q*).

d) L(*Q* <repeat factor>)

is the set of all strings that can be constructed by concatenating *NS*, $LV \le NS \le HV$, strings from L(*Q*).

e) L(<character specifier>)

is a set that contains a single string of length 1 (one) with the character that the <character specifier> represents

f) L(<percent>)

is the set of all strings of any length (zero or more) from the character set of the pattern *PCV*.

g) L(<question mark>)

is the set of all strings that can be constructed by concatenating exactly 0 (zero) or 1 (one) strings from L(*Q*).

h) L(<left paren> *Q* <right paren>)

is equal to L(*Q*)

i) L(<underscore>)

is the set of all strings of length 1 (one) from the character set of the pattern *PCV*.

j) L(<left bracket> <character enumeration> <right bracket>)

is the set of all strings of length 1 (one) from the set of characters in the <character enumeration>s.

k) L(<left bracket> <circumflex> <character enumeration> <right bracket>)

is the set of all strings of length 1 (one) with characters from the character set of the pattern *PCV* that are not contained in the set of characters in the <character enumeration>.

l) L(<left bracket> <character enumeration include> <circumflex> <character enumeration exclude> <right bracket>)

is the set of all strings of length 1 (one) taken from the set of characters in the <character enumeration include>s, except for those strings of length 1 (one) taken from the set of characters in the <character enumeration exclude>.

m) L(<left bracket> <colon> ALPHA <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin letter>s.

n) L(<left bracket> <colon> UPPER <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin upper case letter>s.

o) L(<left bracket> <colon> LOWER <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin lower case letter>s.

p) L(<left bracket> <colon> DIGIT <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <digit>s.

q) L(<left bracket> <colon> SPACE <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are the <space> character.

r) L(<left bracket> <colon> WHITESPACE <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are white space characters.

NOTE 207 — “white space” is defined in Subclause 3.1.6, “Definitions provided in Part 2”.

s) L(<left bracket> <colon> ALNUM <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin letter>s or <digit>s.

t) `L( Q1 || Q2 )`

is the set of all strings that can be constructed by concatenating one element of L(*Q1*) and one element of L(*Q2*).

u) L(*Q*)

is the set of the zero-length string, if *Q* is an empty regular expression.

8) The <similar predicate>

```
CM SIMILAR TO SP
```

is *True*, if there exists at least one element *X* of L(*R*) that is equal to *MCV* according to the collation of the <similar predicate>; otherwise, it is *False*.

NOTE 208 — The <similar predicate> is defined differently from equivalent forms of the LIKE predicate. In particular, blanks at the end of a pattern and collation are handled differently.

Conformance Rules

1) Without Feature T141, “SIMILAR predicate”, conforming SQL language shall not contain a <similar predicate>.

2) Without Feature T042, “Extended LOB data type support”, in conforming SQL language, a <character value expression> simply contained in a <similar predicate> shall not be of declared type CHARACTER LARGE OBJECT.

8.7 <regex like predicate>

Function

Specify a pattern-match comparison using an XQuery regular expression.

Format

```
<regex like predicate> ::=
  <row value predicand> <regex like predicate part 2>

<regex like predicate part 2> ::=
  [ NOT ] LIKE_REGEX <XQuery pattern> [ FLAG <XQuery option flag> ]
```

Syntax Rules

1) The <row value predicand> immediately contained in <regex like predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE* whose declared type is character string.

2) If <XQuery option flag> is not specified, then the zero-length character string is implicit.

3) Let *PAT* be the <XQuery pattern> and let *FL* be the implicit or explicit <XQuery option flag>.

   CVE NOT LIKE_REGEX *PAT* FLAG *FL*

   is equivalent to

   NOT (*CVE* LIKE_REGEX *PAT* FLAG *FL*)

Access Rules

None.

General Rules

1) The value of the <predicate>

   CVE NOT LIKE_REGEX *PAT* FLAG *FL*

   is

   Case:

   a) If the value of *CVE*, *PAT*, or *FL* is the null value, then *Unknown*.

   b) If the list of match vectors that is the result of applying the General Rules of Subclause 9.16, "XQuery regular expression matching", with *STR* as *STRING*, *PAT* as *PATTERN*, 1 (one) as *POSITION*, CHARACTERS as *UNITS*, and *FL* as *FLAG* is non-empty, then *True*.

   c) Otherwise, *False*.

Conformance Rules

1) Without Feature F841, “LIKE_REGEX predicate”, conforming SQL language shall not contain <regex like predicate>.

2) Without Feature F847, “Nonconstant regular expressions”, in conforming SQL language, <XQuery pattern> and <XQuery option flag> shall be <value specification>s.

8.8 <null predicate>

Function

Specify a test for a null value.

Format

```
<null predicate> ::=
  <row value predicand> <null predicate part 2>

<null predicate part 2> ::=
  IS [ NOT ] NULL
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *R* be the value of the <row value predicand>.

2) Case:

   a) If *R* is the null value, then "*R* `IS NULL`" is *True*.

   b) Otherwise:

      i) The value of "*R* `IS NULL`" is

         Case:

         1) If the value of every field in *R* is the null value, then *True*.

         2) Otherwise, *False*.

      ii) The value of "*R* `IS NOT NULL`" is

         Case:

         1) If the value of no field in *R* is the null value, then *True*.

         2) Otherwise, *False*.

NOTE 209 — For all *R*, "*R* IS NOT NULL" has the same result as "NOT *R* IS NULL" if and only if *R* is of degree 1. Table 14, "<null predicate> semantics", specifies this behavior.

Table 14 — <null predicate> semantics

Expression	***R* IS NULL**	***R* IS NOT NULL**	**NOT *R* IS NULL**	**NOT *R* IS NOT NULL**
degree 1: null	*True*	*False*	*False*	*True*
degree 1: not null	*False*	*True*	*True*	*False*
degree > 1: all null	*True*	*False*	*False*	*True*
degree > 1: some null	*False*	*False*	*True*	*True*
degree > 1: none null	*False*	*True*	*True*	*False*

Conformance Rules

None.

8.9 <quantified comparison predicate>

Function

Specify a quantified comparison.

Format

```
<quantified comparison predicate> ::=
  <row value predicand> <quantified comparison predicate part 2>

<quantified comparison predicate part 2> ::=
  <comp op> <quantifier> <table subquery>

<quantifier> ::=
    <all>
  | <some>

<all> ::=
  ALL

<some> ::=
    SOME
  | ANY
```

Syntax Rules

1) Let *RV1* and *RV2* be <row value predicand>s whose declared types are respectively that of the <row value predicand> and the row type of the <table subquery>. The Syntax Rules of Subclause 8.2, "<comparison predicate>", are applied to:

   RV1 <comp op> *RV2*

Access Rules

None.

General Rules

1) Let *R* be the result of the <row value predicand> and let *T* be the result of the <table subquery>.

2) The result of "*R* <comp op> <quantifier> *T*" is derived by the application of the implied <comparison predicate> "*R* <comp op> *RT*" to every row *RT* in *T*.

   Case:

   a) If *T* is empty or if the implied <comparison predicate> is *True* for every row *RT* in *T*, then "*R* <comp op> <all> *T*" is *True*.

   b) If the implied <comparison predicate> is *False* for at least one row *RT* in *T*, then "*R* <comp op> <all> *T*" is *False*.

c) If the implied <comparison predicate> is *True* for at least one row *RT* in *T*, then "*R* <comp op> <some> *T*" is *True*.

d) If *T* is empty or if the implied <comparison predicate> is *False* for every row *RT* in *T*, then "*R* <comp op> <some> *T*" is *False*.

e) If "*R* <comp op> <quantifier> *T*" is neither *True* nor *False*, then it is *Unknown*.

Conformance Rules

None.

NOTE 210 — If <equals operator> or <not equals operator> is specified, then the <quantified comparison predicate> is an equality operator and the Conformance Rules of Subclause 9.9, "Equality operations", apply. Otherwise, the <quantified comparison predicate> is an ordering operation, and the Conformance Rules of Subclause 9.12, "Ordering operations", apply.

8.10 <exists predicate>

Function

Specify a test for a non-empty set.

Format

```
<exists predicate> ::=
  EXISTS <table subquery>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *T* be the result of the <table subquery>.

2) If the cardinality of *T* is greater than 0 (zero), then the result of the <exists predicate> is *True*; otherwise, the result of the <exists predicate> is *False*.

Conformance Rules

1) Without Feature T501, “Enhanced EXISTS predicate”, conforming SQL language shall not contain an <exists predicate> that simply contains a <table subquery> in which the <select list> of a <query specification> directly contained in the <table subquery> does not comprise either an <asterisk> or a single <derived column>.

8.11 <unique predicate>

Function

Specify a test for the absence of duplicate rows.

Format

```
<unique predicate> ::=
  UNIQUE <table subquery>
```

Syntax Rules

1) Each column of user-defined type in the result of the <table subquery> shall have a comparison type.

2) Each column of the <table subquery> is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

Access Rules

None.

General Rules

1) Let *T* be the result of the <table subquery>.

2) If there are no two rows in *T* such that the value of each column in one row is non-null and is not distinct from the value of the corresponding column in the other row, then the result of the <unique predicate> is *True*; otherwise, the result of the <unique predicate> is *False*.

Conformance Rules

1) Without Feature F291, "UNIQUE predicate", conforming SQL language shall not contain a <unique predicate>.

NOTE 211 — The Conformance Rules of Subclause 9.10, "Grouping operations", also apply.

8.12 <normalized predicate>

Function

Determine whether a character string value is normalized.

Format

```
<normalized predicate> ::=
  <row value predicand> <normalized predicate part 2>

<normalized predicate part 2> ::=
  IS [ NOT ] [ <normal form> ] NORMALIZED
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared type of *CVE* shal be character string and the character set of *CVE* shall be UTF8, UTF16, or UTF32.

2) Case:

   a) If <normal form> is specified, then let *NF* be <normal form>.

   b) Otherwise, let *NF* be NFC.

3) The expression

   CVE IS NOT *NF* NORMALIZED

   is equivalent to

   NOT (*CVE* IS *NF* NORMALIZED)

Access Rules

None.

General Rules

1) The result of *CVE* IS *NF* NORMALIZED is

   Case:

   a) If the value of *CVE* is the null value, then *Unknown*.

   b) If the value of *CVE* is in the normalization form specified by *NF*, as defined by [Unicode15], then *True*.

   c) Otherwise, *False*.

Conformance Rules

1) Without Feature T061, “UCS support”, conforming SQL language shall not contain a <normalized predicate>.

2) Without Feature F394, “Optional normal form specification”, conforming SQL language shall not contain <normal form>.

8.13 <match predicate>

Function

Specify a test for matching rows.

Format

```
<match predicate> ::=
  <row value predicand> <match predicate part 2>

<match predicate part 2> ::=
  MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ] <table subquery>
```

Syntax Rules

1) The row type of the <row value predicand> and the row type of the <table subquery> shall be comparable.

2) Each field of <row value predicand> and each column of <table subquery> is an operand of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

3) If neither SIMPLE, PARTIAL, nor FULL is specified, then SIMPLE is implicit.

Access Rules

None.

General Rules

1) Let *R* be the <row value predicand>.

2) If SIMPLE is specified or implicit, then

   Case:

   a) If *R* is the null value, then the <match predicate> is *True*.

   b) Otherwise:

      i) If the value of some field in *R* is the null value, then the <match predicate> is *True*.

      ii) If the value of no field in *R* is the null value, then

         Case:

         1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that

            $R = RT_i$

            then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that

$$R = RT_i$$

then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

3) If PARTIAL is specified, then

Case:

a) If R is the null value, then the <match predicate> is *True*.

b) Otherwise,

Case:

i) If the value of every field in R is the null value, then the <match predicate> is *True*.

ii) Otherwise,

Case:

1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that each non-null value of R equals its corresponding value in RT_i, then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that each non-null value of R equals its corresponding value in RT_i, then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

4) If FULL is specified, then

Case:

a) If R is the null value, then the <match predicate> is *True*.

b) Otherwise,

Case:

i) If the value of every field in R is the null value, then the <match predicate> is *True*.

ii) If the value of no field in R is the null value, then

Case:

1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that

$$R = RT_i$$

then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that

$$R = RT_i$$

then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

iii) Otherwise, the <match predicate> is *False*.

Conformance Rules

1) Without Feature F741, “Referential MATCH types”, conforming SQL language shall not contain a <match predicate>.

NOTE 212 — The Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

8.14 <overlaps predicate>

Function

Specify a test for an overlap between two datetime periods.

Format

```
<overlaps predicate> ::=
  <overlaps predicate part 1> <overlaps predicate part 2>

<overlaps predicate part 1> ::=
  <row value predicand 1>

<overlaps predicate part 2> ::=
  OVERLAPS <row value predicand 2>

<row value predicand 1> ::=
  <row value predicand>

<row value predicand 2> ::=
  <row value predicand>
```

Syntax Rules

1) The degrees of <row value predicand 1> and <row value predicand 2> shall both be 2.

2) The declared types of the first field of <row value predicand 1> and the first field of <row value predicand 2> shall both be datetime data types and these data types shall be comparable.

   NOTE 213 — Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.6.2, "Datetimes".

3) The declared type of the second field of each <row value predicand> shall be a datetime data type or INTERVAL.

   Case:

   a) If the declared type is INTERVAL, then the precision of the declared type shall be such that the interval can be added to the datetime data type of the first column of the <row value predicand>.

   b) If the declared type is a datetime data type, then it shall be comparable with the datetime data type of the first column of the <row value predicand>.

Access Rules

None.

General Rules

1) If the value of <row value predicand 1> is the null value or the value of <row value predicand 2> is the null value, then the result of the <overlaps predicate> is *Unknown* and no further General Rules of this Subclause are applied.

2) Let *D1* be the value of the first field of <row value predicand 1> and *D2* be the value of the first field of <row value predicand 2>.

3) Case:

   a) If the most specific type of the second field of <row value predicand 1> is a datetime data type, then let *E1* be the value of the second field of <row value predicand 1>.

   b) If the most specific type of the second field of <row value predicand 1> is INTERVAL, then let *I1* be the value of the second field of <row value predicand 1>. Let *E1* = *D1* + *I1*.

4) If *D1* is the null value or if *E1* < *D1*, then let *S1* = *E1* and let *T1* = *D1*. Otherwise, let *S1* = *D1* and let *T1* = *E1*.

5) Case:

   a) If the most specific type of the second field of <row value predicand 2> is a datetime data type, then let *E2* be the value of the second field of <row value predicand 2>.

   b) If the most specific type of the second field of <row value predicand 2> is INTERVAL, then let *I2* be the value of the second field of <row value predicand 2>. Let *E2* = *D2* + *I2*.

6) If *D2* is the null value or if *E2* < *D2*, then let *S2* = *E2* and let *T2* = *D2*. Otherwise, let *S2* = *D2* and let *T2* = *E2*.

7) The result of the <overlaps predicate> is the result of the following expression:

```
( S1 > S2 AND NOT ( S1 >= T2 AND T1 >= T2 ) )
     OR
( S2 > S1 AND NOT ( S2 >= T1 AND T2 >= T1 ) )
     OR
( S1 = S2 AND ( T1 <> T2 OR T1 = T2 ) )
```

Conformance Rules

1) Without Feature F053, “OVERLAPS predicate”, conforming SQL language shall not contain an <overlaps predicate>.

8.15 <distinct predicate>

Function

Specify a test of whether two row values are distinct

Format

```
<distinct predicate> ::=
  <row value predicand 3> <distinct predicate part 2>

<distinct predicate part 2> ::=
  IS [ NOT ] DISTINCT FROM <row value predicand 4>

<row value predicand 3> ::=
  <row value predicand>

<row value predicand 4> ::=
  <row value predicand>
```

Syntax Rules

1) The two <row value predicand>s shall be of the same degree.

2) Let *respective values* be values with the same ordinal position.

3) The declared types of the respective values of the two <row value predicand>s shall be comparable.

4) Let *X* be the first <row value predicand> and let *Y* be the second <row value predicand>.

5) Each field of each <row value predicand> is an operand of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

6) If <distinct predicate part 2> immediately contains NOT, then the <distinct predicate> is equivalent to:

   `NOT ( X IS DISTINCT FROM Y )`

Access Rules

None.

General Rules

1) The result of <distinct predicate> is *True* if the value of <row value predicand 3> is distinct from the value of <row value predicand 4>; otherwise, the result is *False*.

   NOTE 214 — "distinct" is defined in Subclause 3.1.6, "Definitions provided in Part 2".

2) If two <row value predicand>s are not distinct, then they are said to be *duplicates*. If a number of <row value predicand>s are all duplicates of each other, then all except one are said to be *redundant duplicates*.

Conformance Rules

1) Without Feature T151, “DISTINCT predicate”, conforming SQL language shall not contain a <distinct predicate>.

   NOTE 215 — The Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

2) Without Feature T152, “DISTINCT predicate with negation”, conforming SQL language shall not contain a <distinct predicate part 2> that immediately contains NOT.

8.16 <member predicate>

Function

Specify a test of whether a value is a member of a multiset.

Format

```
<member predicate> ::=
  <row value predicand> <member predicate part 2>

<member predicate part 2> ::=
  [ NOT ] MEMBER [ OF ] <multiset value expression>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression> and let *ET* be the declared element type of *MVE*.

2) Case:

   a) If the <row value predicand> is a <row value constructor predicand> that is a single <common value expression> or <boolean value expression> *CVE*, then let *X* be *CVE*.

   b) Otherwise, let *X* be the <row value predicand>.

3) The declared type of *X* shall be comparable to *ET*.

4) *X* is an operand of an equality operation. The Syntax Rules of Subclause 9.9, "Equality operations", apply.

5) If <member predicate part 2> immediately contains NOT, then the <member predicate> is equivalent to

   `NOT ( X MEMBER OF MVE )`

Access Rules

None.

General Rules

1) Let *XV* be the value of *X*, and let *MV* be the value of *MVE*.

2) Let *N* be the result of `CARDINALITY (MVE)`.

3) The <member predicate>

   `XV MEMBER OF MVE`

   is evaluated as follows.

   Case:

   a) If *N* is 0 (zero), then the <member predicate> is *False*.

b) If either *XV* or *MV* is the null value, then the <member predicate> is *Unknown*.

c) Otherwise, let ME_i for 1 (one) ≤ i ≤ N be an enumeration of the elements of *MV*.

Case:

i) If $CV = ME_i$ for some i, then the <member predicate> is *True*.

ii) If ME_i is the null value for some i, then the <member predicate> is *Unknown*.

iii) Otherwise, the <member predicate> is *False*.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <member predicate>.

NOTE 216 — The Conformance Rules of Subclause 9.9, "Equality operations", also apply.

8.17 <submultiset predicate>

Function

Specify a test of whether a multiset is a submultiset of another multiset.

Format

```
<submultiset predicate> ::=
  <row value predicand> <submultiset predicate part 2>

<submultiset predicate part 2> ::=
  [ NOT ] SUBMULTISET [ OF ] <multiset value expression>
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor> that is a single <common value expression> *CVE*. The declared type of *CVE* shall be a multiset type. Let *CVET* be the declared element type of *CVE*.

2) Let *MVE* be the <multiset value expression>. Let *MVET* be the declared element type of *MVE*.

3) *CVET* shall be comparable to *MVET*.

4) *CVE* and *MVE* are multiset operands of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

5) If <submultiset predicate part 2> immediately contains NOT, then the <member predicate> is equivalent to

   NOT (*CVE* SUBMULTISET OF *MVE*)

Access Rules

None.

General Rules

1) Let *CV* be the value of *CVE*, and let *MV* be the value of *MVE*.

2) Let *M* be the result of CARDINALITY (*CV*), and let *N* be the result of CARDINALITY (*MV*).

3) The <submultiset predicate>

   CVE SUBMULTISET OF *MVE*

   is evaluated as follows.

   Case:

   a) If *M* is 0 (zero), then the <submultiset predicate> is *True*.

b) If either *CV* or *MV* is the null value, then the <submultiset predicate> is *Unknown*.

c) Otherwise,

Case:

i) If $M > N$, then the <submultiset predicate> is *False*.

ii) If there exist an enumeration CE_i for 1 (one) $\leq i \leq M$ of the elements of *CE* and an enumeration ME_j for 1 (one) $\leq j \leq N$ of the elements of *MV* such that for all i, 1 (one) $\leq i \leq M$, $CE_i = ME_i$, then the <submultiset predicate> is *True*.

iii) If there exist an enumeration CE_i for 1 (one) $\leq i \leq M$ of the elements of *CE* and an enumeration ME_i for 1 (one) $\leq i \leq N$ of the elements of *MV* such that for all i, 1 (one) $\leq i \leq M$, $CE_i = ME_i$ is either *True* or *Unknown*, then the <submultiset predicate> is *Unknown*.

iv) Otherwise, the <submultiset predicate> is *False*.

Conformance Rules

1) Without Feature S275, “Advanced multiset support”, conforming SQL language shall not contain a <submultiset predicate>.

NOTE 217 — The Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

8.18 <set predicate>

Function

Specify a test of whether a multiset is a set (that is, does not contain any duplicates).

Format

```
<set predicate> ::=
  <row value predicand> <set predicate part 2>

<set predicate part 2> ::=
  IS [ NOT ] A SET
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a single <common value expression> *CVE*. The declared type of *CVE* shall be a multiset type. Let *CVET* be the element type of *CVE*.

2) *CVE* is an operand of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply.

3) If <set predicate part 2> immediately contains NOT, then the <set predicate> is equivalent to

   ```
   NOT ( CVE IS A SET )
   ```

4) If <set predicate part 2> does not immediately contain NOT, then the <set predicate> is equivalent to

   ```
   CARDINALITY ( CVE ) = CARDINALITY ( SET ( CVE ) )
   ```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <set predicate>.

   NOTE 218 — The Conformance Rules of Subclause 9.11, "Multiset element grouping operations", also apply.

8.19 <type predicate>

Function

Specify a type test.

Format

```
<type predicate> ::=
  <row value predicand> <type predicate part 2>

<type predicate part 2> ::=
  IS [ NOT ] OF <left paren> <type list> <right paren>

<type list> ::=
  <user-defined type specification>
      [ { <comma> <user-defined type specification> }... ]

<user-defined type specification> ::=
    <inclusive user-defined type specification>
  | <exclusive user-defined type specification>

<inclusive user-defined type specification> ::=
  <path-resolved user-defined type name>

<exclusive user-defined type specification> ::=
  ONLY <path-resolved user-defined type name>
```

Syntax Rules

1) The <row value predicand> immediately contained in <type predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE*.

2) The declared type of *CVE* shall be a user-defined type.

3) For each <user-defined type name> *UDTN* contained in a <user-defined type specification>, the schema identified by the implicit or explicit schema name of *UDTN* shall include a user-defined type descriptor whose name is equivalent to the <qualified identifier> of *UDTN*.

4) Let the term *specified type* refer to a user-defined type that is specified by a <user-defined type name> contained in a <user-defined type specification>. A type specified by an <inclusive user-defined type specification> is *inclusively specified*; a type specified by an <exclusive user-defined type specification> is *exclusively specified*.

5) Let *T* be the type specified by <inclusive user-defined type specification> or <exclusive user-defined type specification>. *T* shall be a subtype of the declared type of *CVE*.

   NOTE 219 — The term "subtype family" is defined in Subclause 4.7.5, "Subtypes and supertypes". If *T1* is a member of the subtype family of *T2*, then it follows that the subtype family of *T1* and the subtype family of *T2* are the same set of types.

6) Let *TL* be the <type list>.

7) A <type predicate> of the form

```
CVE IS NOT
OF (TL)
```

is equivalent to

```
NOT ( CVE IS
OF (TL) )
```

Access Rules

None.

General Rules

1) Let *V* be the result of evaluating the <row value predicand>.

2) Let *ST* be the set consisting of every type that is either some exclusively specified type, or a subtype of some inclusively specified type.

3) Let *TPR* be the result of evaluating the <type predicate>.

   Case:

   a) If *V* is the null value, then *TPR* is *Unknown*.

   b) If the most specific type of *V* is a member of *ST*, then *TPR* is *True*.

   c) Otherwise, *TPR* is *False*.

Conformance Rules

1) Without Feature S151, “Type predicate”, conforming SQL language shall not contain a <type predicate>.

8.20 <search condition>

Function

Specify a condition that is *True*, *False*, or *Unknown*, depending on the value of a <boolean value expression>.

Format

```
<search condition> ::=
  <boolean value expression>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) The result of the <search condition> is the result of the <boolean value expression>.

Conformance Rules

None.

(Blank page)

9 Additional common rules

This Clause is modified by Clause 9, "Additional common rules", in ISO/IEC 9075-9.
This Clause is modified by Clause 10, "Additional common rules", in ISO/IEC 9075-14.

9.1 Retrieval assignment

This Subclause is modified by Subclause 9.1, "Retrieval assignment", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.1, "Retrieval assignment", in ISO/IEC 9075-14.

Function

Specify rules for assignments to targets that do not support null values or that support null values with indicator parameters (*e.g.*, assigning SQL-data to host parameters or host variables).

Syntax Rules

1) Let *T* be the *TARGET* and let and *V* be the *VALUE* specified in an application of this Subclause. Let the declared types of *T* and *V* be *TD* and *SD*, respectively.

2) If *TD* is numeric, boolean, datetime, interval, or a user-defined type, then either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

   NOTE 220 — "Appropriate user-defined cast function" is defined in Subclause 4.11, "Data conversions".

3) If *TD* is binary string, then

   Case:

   a) If *T* is either a locator parameter of an external routine, a locator variable, or a host parameter that is a binary large object locator parameter, then *SD* shall be BINARY LARGE OBJECT and *SD* shall be assignable to *TD*.

   b) Otherwise, either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

4) If *TD* is character string, then

   Case:

   a) If *T* is either a locator parameter of an external routine, a locator variable, or a host parameter that is a character large object locator parameter, then *SD* shall be CHARACTER LARGE OBJECT and *SD* shall be assignable to *TD*.

   b) Otherwise, either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

5) If the declared type of *T* is a reference type, then the declared type of *V* shall be a reference type whose referenced type is a subtype of the referenced type of *T*.

6) If the declared type of *T* is a row type, then:

   a) The declared type of *V* shall be a row type.

   b) The degree of *V* shall be the same as the degree of *T*. Let *n* be that degree.

   c) Let TT_i, 1 (one) ≤ i ≤ n, be the declared type of the *i*-th field of *T*, let VT_i be the declared type of the *i*-th field of *V*, let $T1_i$ be an arbitrary target whose declared type is TT_i, and let $V1_i$ be an arbitrary expression whose declared type is VT_i. For each *i*, 1 (one) ≤ i ≤ n, the Syntax Rules of this Subclause apply to T_i V_i, as *TARGET* and *VALUE*, respectively.

7) If the declared type of *T* is a collection type, then:

   a) If the declared type of *T* is an array type, then the declared type of *V* shall be an array type.

   b) If the declared type of *T* is a multiset type, then the declared type of *V* shall be a multiset type.

   c) Let *TT* be the element type of the declared type of *T*, let *VT* be the element type of the declared type of *V*, let *T1* be an arbitrary target whose declared type is *TT*, and let *V1* be an arbitrary expression whose declared type is *VT*. The Syntax Rules of this Subclause apply to *T1* and *V1*, as *TARGET* and *VALUE*, respectively.

Access Rules

None.

General Rules

1) If the declared type of *V* is not assignable to the declared type of *T*, then for the remaining General Rules of this Subclause *V* is effectively replaced by the result of evaluating the expression *UDCF*(*V*).

2) If *V* is the null value and *T* is a host parameter, then

   Case:

   a) If an indicator parameter is specified for *T*, then that indicator parameter is set to –1.

   b) If no indicator parameter is specified for *T*, then an exception condition is raised: *data exception — null value, no indicator parameter*.

3) If *V* is the null value and *T* is a host variable, then

   Case:

   a) If an indicator variable is specified for *T*, then that indicator variable is set to –1.

   b) If no indicator variable is specified for *T*, then an exception condition is raised: *data exception — null value, no indicator parameter*.

4) If *V* is not the null value, *T* is a host parameter, and *T* has an indicator parameter, then

   Case:

a) If the declared type of *T* is character string or binary string and the length *M* in characters or octets, respectively, of *V* is greater than the length in characters or octets, respectively, of *T*, then the indicator parameter is set to *M*. If *M* exceeds the maximum value that the indicator parameter can contain, then an exception condition is raised: *data exception — indicator overflow*.

b) Otherwise, the indicator parameter is set to 0 (zero).

5) If *V* is not the null value, *T* is a host variable, and *T* has an indicator variable, then

Case:

a) If the declared type of *T* is character string or binary string and the length in characters or octets, respectively, *M* of *V* is greater than the length in characters or octets, respectively, of *T*, then the indicator parameter is set to *M*. If *M* exceeds the maximum value that the indicator parameter can contain, then an exception condition is raised: *data exception — indicator overflow*.

b) Otherwise, the indicator variable is set to 0 (zero).

6) If *V* is not the null value, then

Case:

a) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters of *V* is equal to *L*, then the value of *T* is set to *V*.

b) If the declared type of *T* is fixed-length character string with length in characters *L*, and the length in characters of *V* is greater than *L*, then the value of *T* is set to the first *L* characters of *V* and a completion condition is raised: *warning — string data, right truncation*.

c) If the declared type of *T* is fixed-length character string with length in characters *L*, and the length in characters *M* of *V* is smaller than *L*, then the first *M* characters of *T* are set to *V*, and the last *L–M* characters of *T* are set to <space>s.

d) If the declared type of *T* is variable-length character string and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

e) If the declared type of *T* is variable-length character string and the length in characters of *V* is greater than the maximum length in characters *L* of *T*, then the value of *T* is set to the first *L* characters of *V*, then the length in characters of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation*.

f) If the declared type of *T* is a character large object type and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

g) If the declared type of *T* is a character large object type and the length in characters of *V* is greater than the maximum length in characters *L* of *T*, then the value of *T* is set to the first *L* characters of *V*, the length in characters of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation*.

h) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets of *V* is equal to *L*, then the value of *T* is set to *V*.

i) If the declared type of *T* is fixed-length binary string with length in octets *L*, and the length in octets of *V* is greater than *L*, then the value of *T* is set to the first *L* octets of *V* and a completion condition is raised: *warning — string data, right truncation*.

j) If the declared type of *T* is fixed-length binary string with length in octets *L*, and the length in octets *M* of *V* is less than *L*, then the first *M* octets of *T* are set to *V*, and the last *L–M* octets of *T* are set to X'00's.

k) If the declared type of *T* is variable-length binary string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

l) If the declared type of *T* is variable-length binary string and the length in octets of *V* is greater than the maximum length in octets *L* of *T*, then the value of *T* is set to the first *L* octets of *V*, then the length in octets of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation*.

m) If the declared type of *T* is binary large object string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

n) If the declared type of *T* is binary large object string and the length in octets of *V* is greater than the maximum length in octets *L* of *T*, then the value of *T* is set to the first *L* octets of *V*, the length in octets of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation*.

o) If the declared type of *T* is numeric, then

Case:

i) If *V* is a member of the declared type of *T*, then *T* is set to *V*.

ii) If a member of the declared type of *T* can be obtained from *V* by rounding or truncation, then *T* is set to that value. If the declared type of *T* is exact numeric, then it is implementation-defined whether the approximation is obtained by rounding or by truncation.

iii) Otherwise, an exception condition is raised: *data exception — numeric value out of range*.

p) If the declared type of *T* is boolean, then the value of *T* is set to *V*.

q) If the declared type *DT* of *T* is datetime, then:

i) If only one of *DT* and the declared type of *V* is datetime with time zone, then *V* is effectively replaced by

```
CAST ( V AS DT )
```

ii) Case:

1) If *V* is a member of the declared type of *T*, then *T* is set to *V*.

2) If a member of the declared type of *T* can be obtained from *V* by rounding or truncation, then *T* is set to that value. It is implementation-defined whether the approximation is obtained by rounding or truncation.

3) Otherwise, an exception condition is raised: *data exception — datetime field overflow*.

r) If the declared type of *T* is interval, then

Case:

i) If *V* is a member of the declared type of *T*, then *T* is set to *V*.

ii) If a member of the declared type of *T* can be obtained from *V* by rounding or truncation, then *T* is set to that value. It is implementation-defined whether the approximation is obtained by rounding or by truncation.

iii) Otherwise, an exception condition is raised: *data exception — interval field overflow*.

s) If the declared type of *T* is a row type, then:

i) Let *n* be the degree of *T*.

ii) For *i* ranging from 1 (one) to *n*, the General Rules of this Subclause are applied to the *i*-th element of *T* and the *i*-th element of *V* as *TARGET* and *VALUE*, respectively.

t) If the declared type of *T* is a user-defined type, then the value of *T* is set to *V*.

u) If the declared type of *T* is a reference type, then the value of *T* is set to *V*.

v) If the declared type of *T* is an array type, then

Case:

i) If the maximum cardinality *L* of *T* is equal to the cardinality *M* of *V*, then the elements of *T* are set to the values of the corresponding elements of *V* by applying the General Rules of this Subclause to each pair of elements with the element of *T* as *TARGET* and the element of *V* as *VALUE*.

ii) If the maximum cardinality *L* of *T* is smaller than the cardinality *M* of *V*, then the elements of *T* are set to the values of the first *L* corresponding elements of *V* by applying the General Rules of this Subclause to each pair of elements with the element of *T* as *TARGET* and the element of *V* as *VALUE*; a completion condition is raised: *warning — array data, right truncation*.

iii) If the maximum cardinality *L* of *T* is greater than the cardinality *M* of *V*, then the *M* first elements of *T* are set to the values of the corresponding elements of *V* by applying the General Rules of this Subclause to each pair of elements with the element of *T* as *TARGET* and the element of *V* as *VALUE*. The cardinality of the value of *T* is *M*.

NOTE 221 — The maximum cardinality *L* of *T* is unchanged.

w) If the declared type of *T* is a multiset type, then the value of *T* is set to *V*.

Conformance Rules

None.

9.2 Store assignment

This Subclause is modified by Subclause 9.2, "Store assignment", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.2, "Store assignment", in ISO/IEC 9075-14.

Function

Specify rules for assignments where the target permits null without the use of indicator parameters or indicator variables, such as storing SQL-data or setting the value of SQL parameters.

Syntax Rules

1) Let *T* and *V* be the *TARGET* and *VALUE* specified in an application of this Subclause. Let the declared types of *T* and *V* be *TD* and *SD*, respectively.

2) If *TD* is character string, binary string, numeric, boolean, datetime, interval, or a user-defined type, then either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

   NOTE 222 — "Appropriate user-defined cast function" is defined in Subclause 4.11, "Data conversions".

3) If the declared type of *T* is a reference type, then the declared type of *V* shall be a reference type whose referenced type is a subtype of the referenced type of *T*.

4) If the declared type of *T* is a row type, then:

   a) The declared type of *V* shall be a row type.

   b) The degree of *V* shall be the same as the degree of *T*. Let *n* be that degree.

   c) Let TT_i, 1 (one) ≤ i ≤ n, be the declared type of the i-th field of *T*, let VT_i be the declared type of the i-th field of *V*, let $T1_i$ be an arbitrary target whose declared type is TT_i, and let $V1_i$ be an arbitrary expression whose declared type is VT_i. For each i, 1 (one) ≤ i ≤ n, the Syntax Rules of this Subclause apply to T_i V_i, as *TARGET* and *VALUE*, respectively.

5) If the declared type of *T* is a collection type, then:

   a) If the declared type of *T* is an array type, then the declared type of *V* shall be an array type.

   b) If the declared type of *T* is a multiset type, then the declared type of *V* shall be a multiset type.

   c) Let *TT* be the element type of the declared type of *T*, let *VT* be the element type of the declared type of *V*, let *T1* be an arbitrary target whose declared type is *TT*, and let *V1* be an arbitrary expression whose declared type is *VT*. The Syntax Rules of this Subclause apply to *T1* and *V1*, as *TARGET* and *VALUE*, respectively.

Access Rules

None.

General Rules

1) If the declared type of *V* is not assignable to the declared type of *T*, then for the remaining General Rules of this Subclause *V* is effectively replaced by the result of evaluating the expression *UDCF*(*V*).

2) Case:

   a) If *V* is the null value, then

      Case:

      i) If *V* is specified using NULL, then *T* is set to the null value.

      ii) If *V* is a host parameter and contains an indicator parameter, then

         Case:

         1) If the value of the indicator parameter is equal to –1, then *T* is set to the null value.

         2) If the value of the indicator parameter is less than –1, then an exception condition is raised: *data exception — invalid indicator parameter value*.

      iii) If *V* is a host variable and contains an indicator variable, then

         Case:

         1) If the value of the indicator variable is equal to –1, then *T* is set to the null value.

         2) If the value of the indicator variable is less than –1, then an exception condition is raised: *data exception — invalid indicator parameter value*.

      iv) Otherwise, *T* is set to the null value.

   b) Otherwise,

      Case:

      i) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters of *V* is equal to *L*, then the value of *T* is set to *V*.

      ii) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters *M* of *V* is larger than *L*, then

         Case:

         1) If the rightmost *M–L* characters of *V* are all <space>s, then the value of *T* is set to the first *L* characters of *V*.

         2) If one or more of the rightmost *M–L* characters of *V* are not <space>s, then an exception condition is raised: *data exception — string data, right truncation*.

      iii) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters *M* of *V* is less than *L*, then the first *M* characters of *T* are set to *V* and the last *L–M* characters of *T* are set to <space>s.

      iv) If the declared type of *T* is variable-length character string and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

v) If the declared type of *T* is variable-length character string and the length in characters *M* of *V* is greater than the maximum length in characters *L* of *T*, then

Case:

1) If the rightmost *M–L* characters of *V* are all <space>s, then the value of *T* is set to the first *L* characters of *V* and the length in characters of *T* is set to *L*.

2) If one or more of the rightmost *M–L* characters of *V* are not <space>s, then an exception condition is raised: *data exception — string data, right truncation*.

vi) If the declared type of *T* is a character large object type and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

vii) If the declared type of *T* is a character large object type and the length in characters *M* of *V* is greater than the maximum length in characters *L* of *T*, then

Case:

1) If the rightmost *M–L* characters of *V* are all <space>s, then the value of *T* is set to the first *L* characters of *V* and the length in characters of *T* is set to *L*.

2) If one or more of the rightmost *M–L* characters of *V* are not <space>s, then an exception condition is raised: *data exception — string data, right truncation*.

viii) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets of *V* is equal to *L*, then the value of *T* is set to *V*.

ix) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets *M* of *V* is larger than *L*, then

Case:

1) If the rightmost *M–L* octets of *V* are all equal to X'00', then the value of *T* is set to the first *L* octets of *V*.

2) If one or more of the rightmost *M–L* octets of *V* are not equal to X'00', then an exception condition is raised: *data exception — string data, right truncation*.

x) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets *M* of *V* is less than *L*, then the first *M* octets of *T* are set to *V* and the last *L–M* octets of *T* are set to X'00's.

xi) If the declared type of *T* is variable-length binary string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

xii) If the declared type of *T* is variable-length binary string and the length in octets *M* of *V* is greater than the maximum length in octets *L* of *T*, then

Case:

1) If the rightmost *M–L* octets of *V* are all equal to X'00', then the value of *T* is set to the first *L* octets of *V* and the length in octets of *T* is set to *L*.

2) If one or more of the rightmost *M–L* octets of *V* are not equal to X'00', then an exception condition is raised: *data exception — string data, right truncation*.

xiii) If the declared type of *T* is binary large object string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

xiv) If the declared type of *T* is binary large object string and the length in octets *M* of *V* is greater than the maximum length in octets *L* of *T*, then

Case:

1) If the rightmost *M–L* octets of *V* are all equal to X'00', then the value of *T* is set to the first *L* octets of *V* and the length in octets of *T* is set to *L*.

2) If one or more of the rightmost *M–L* octets of *V* are not equal to X'00', then an exception condition is raised: *data exception — string data, right truncation*.

xv) If the declared type of *T* is numeric, then

Case:

1) If *V* is a member of the declared type of *T*, then *T* is set to *V*.

2) If a member of the declared type of *T* can be obtained from *V* by rounding or truncation, then *T* is set to that value. If the declared type of *T* is exact numeric, then it is implementation-defined whether the approximation is obtained by rounding or by truncation.

3) Otherwise, an exception condition is raised: *data exception — numeric value out of range*.

xvi) If the declared type *DT* of *T* is datetime, then

1) If only one of *DT* and the declared type of *V* is datetime with time zone, then *V* is effectively replaced by

```
CAST ( V AS DT )
```

2) Case:

A) If *V* is a member of the declared type of *T*, then *T* is set to *V*.

B) If a member of the declared type of *T* can be obtained from *V* by rounding or truncation, then *T* is set to that value. It is implementation-defined whether the approximation is obtained by rounding or truncation.

C) Otherwise, an exception condition is raised: *data exception — datetime field overflow*.

xvii) If the declared type of *T* is interval, then

Case:

1) If *V* is a member of the declared type of *T*, then *T* is set to *V*.

2) If a member of the declared type of *T* can be obtained from *V* by rounding or truncation, then *T* is set to that value. It is implementation-defined whether the approximation is obtained by rounding or by truncation.

3) Otherwise, an exception condition is raised: *data exception — interval field overflow*.

xviii) If the declared type of *T* is boolean, then the value of *T* is set to *V*.

xix) If the declared type of *T* is a row type, then:

1) Let *n* be the degree of *T*.

2) For *i* ranging from 1 (one) to *n*, the General Rules of this Subclause are applied to the *i*-th element of *T* and the *i*-th element of *V* as *TARGET* and *VALUE*, respectively.

xx) If the declared type of *T* is a user-defined type, then the value of *T* is set to *V*.

xxi) If the declared type of *T* is a reference type, then the value of *T* is set to *V*.

xxii) If the declared type of *T* is an array type, then

Case:

1) If the maximum cardinality *L* of *T* is equal to the cardinality *M* of *V*, then the elements of *T* are set to the values of the corresponding elements of *V* by applying the General Rules of this Subclause to each pair of elements with the element of *T* as *TARGET* and the element of *V* as *VALUE*.

2) If the maximum cardinality *L* of *T* is smaller than the cardinality *M* of *V*, then

Case:

A) If the rightmost *M–L* elements of *V* are all null, then the elements of *T* are set to the values of the first *L* corresponding elements of *V* by applying the General Rules of this Subclause to each pair of elements with the element of *T* as *TARGET* and the element of *V* as *VALUE*.

B) If one or more of the rightmost *M–L* elements of *V* are not the null value, then an exception condition is raised: *data exception — array data, right truncation*.

3) If the maximum cardinality *L* of *T* is greater than the cardinality *M* of *V*, then the *M* first elements of *T* are set to the values of the corresponding elements of *V* by applying the General Rules of this Subclause to each pair of elements with the element of *T* as *TARGET* and the element of *V* as *VALUE*. The cardinality of the value of *T* is set to *M*.

NOTE 223 — The maximum cardinality *L* of *T* is unchanged.

xxiii) If the declared type of *T* is a multiset type, then the value of *T* is set to *V*.

Conformance Rules

None.

9.3 Result of data type combinations

This Subclause is modified by Subclause 9.3, "Result of data type combinations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.3, "Result of data type combinations", in ISO/IEC 9075-14.

Function

Specify the result data type of the result of certain combinations of values of compatible data types, such as <case expression>s, <collection value expression>s, or a column in the result of a <query expression>.

Syntax Rules

1) Let *IDTS* be the set of data types specified in an application of this Subclause. Let *DTS* be the set of data types in *IDTS* excluding any data types that are undefined. If the cardinality of *DTS* is 0 (zero), then the result data type is undefined and no further Rules of this Subclause are evaluated.

   NOTE 224 — The notion of "undefined data type" is defined in Subclause 20.6, "<prepare statement>".

2) All of the data types in *DTS* shall be comparable.

3) Case:

   a) If any of the data types in *DTS* is character string, then:

      i) All data types in *DTS* shall be character string, and all of them shall have the same character repertoire. The character set of the result is the character set of the data type in *DTS* that has the character encoding form with the higest precedence.

      ii) The collation derivation and declared type collation of the result are determined as follows.

         Case:

         1) If some data type in *DTS* has an *explicit* collation derivation and declared type collation *EC1*, then every data type in *DTS* that has an *explicit* collation derivation shall have a declared type collation that is *EC1*. The collation derivation is *explicit* and the collation is *EC1*.

         2) If every data type in *DTS* has an *implicit* collation derivation, then

            Case:

            A) If every data type in *DTS* has the same declared type collation *IC1*, then the collation derivation is *implicit* and the declared type collation is *IC1*.

            B) Otherwise, the collation derivation is *none*.

         3) Otherwise, the collation derivation is *none*.

      iii) Case:

         1) If any of the data types in *DTS* is a character large object type, then the result data type is a character large object type with maximum length in characters equal to the maximum of the lengths in characters and maximum lengths in characters of the data types in *DTS*.

2) If any of the data types in *DTS* is variable-length character string, then the result data type is variable-length character string with maximum length in characters equal to the maximum of the lengths in characters and maximum lengths in characters of the data types in *DTS*.

3) Otherwise, the result data type is fixed-length character string with length in characters equal to the maximum of the lengths in characters of the data types in *DTS*.

b) If any of the data types in *DTS* is binary string, then

Case:

i) If any of the data types in *DTS* is a binary large object type, then the result data type is a binary large object type with maximum length in octets equal to the maximum of the maximum lengths in octets of the data types in *DTS*.

ii) If any of the data types in *DTS* is variable-length binary string, then the result data type is variable-length binary string with maximum length in octets equal to the maximum of the lengths in octets of the maximum lengths in octets of the data types in *DTS*.

iii) Otherwise, the result data type is fixed-length binary string with length in octets equal to the maximum of the lengths in octets of the data types in *DTS*.

c) If all of the data types in *DTS* are exact numeric, then the result data type is exact numeric with implementation-defined precision and with scale equal to the maximum of the scales of the data types in *DTS*.

d) If any data type in *DTS* is approximate numeric, then each data type in *DTS* shall be numeric and the result data type is approximate numeric with implementation-defined precision.

e) If some data type in *DTS* is a datetime data type, then every data type in *DTS* shall be a datetime data type having the same datetime fields. The result data type is a datetime data type having the same datetime fields, whose fractional seconds precision is the largest of the fractional seconds precisions in *DTS*. If some data type in *DTS* has a time zone displacement value, then the result has a time zone displacement value; otherwise, the result does not have a time zone displacement value.

f) If any data type in *DTS* is interval, then each data type in *DTS* shall be interval. If the precision of any data type in *DTS* specifies YEAR or MONTH, then the precision of each data type shall specify only YEAR or MONTH. If the precision of any data type in *DTS* specifies DAY, HOUR, MINUTE, or SECOND(*N*), then the precision of no data type of *DTS* shall specify the <primary datetime field>s YEAR and MONTH. The result data type is interval with precision "*S* TO *E*", where *S* and *E* are the most significant of the <start field>s and the least significant of the <end field>s of the data types in *DTS*, respectively.

g) If any data type in *DTS* is boolean, then each data type in *DTS* shall be boolean. The result data type is boolean.

h) If any data type in *DTS* is a row type, then each data type in *DTS* shall be a row type with the same degree and the data type of each field in the same ordinal position of every row type shall be comparable. The result data type is a row type defined by an ordered sequence of (<field name>, data type) pairs FD_i, where data type is the data type resulting from the application of this Subclause to the set of data types of fields in the same ordinal position as FD_i in every row type in *DTS* and <field name> is determined as follows.

Case:

i) If the names of fields in the same ordinal position as FD_i in every row type in *DTS* is *F*, then the <field name> in FD_i is *F*.

ii) Otherwise, the <field name> in FD_i is implementation-dependent.

i) If any data type in *DTS* is an array type then every data type in *DTS* shall be an array type. The data type of the result is array type with element data type *ETR*, where *ETR* is the data type resulting from the application of this Subclause to the set of element types of the array types of *DTS*, and maximum cardinality equal to the maximum of the maximum cardinalities of the data types in *DTS*.

j) If any data type in *DTS* is a multiset type then every data type in *DTS* shall be a multiset type. The data type of the result is multiset type with element data type *ETR*, where *ETR* is the data type resulting from the application of this Subclause to the set of element types of the multiset types of *DTS*.

k) If any data type in *DTS* is a reference type, then there shall exist a subtype family *STF* such that each data type in *DTS* is a member of *STF*. Let *RT* be the minimal common supertype of each data type in *DTS*.

Case:

i) If the data type descriptor of every data type in *DTS* includes the name of a referenceable table identifying the scope of the reference type, and every such name is equivalent to some name *STN*, then result data type is:

```
RT SCOPE ( STN )
```

ii) Otherwise, the result data type is *RT*.

l) Otherwise, there shall exist a subtype family *STF* such that each data type in *DTS* is a member of *STF*. The result data type is the minimal common supertype of each data type in *DTS*.

NOTE 225 — *Minimal common supertype* is defined in Subclause 4.7.5, “Subtypes and supertypes”.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.4 Subject routine determination

Function

Determine the subject routine of a given routine invocation.

Syntax Rules

1) Let *SR* and *AL* be respectively the set of SQL-invoked routines, arbitrarily ordered, and the <SQL argument list> specified in an application of this Subclause.

2) Let *n* be the number of SQL-invoked routines in *SR*. Let R_i, 1 (one) ≤ i ≤ n, be the *i*-th SQL-invoked routine in *SR* in the ordering of *SR*.

3) Let *m* be the number of SQL arguments in *AL*. Let A_j, 1 (one) ≤ j ≤ m, be the *j*-th SQL argument in *AL*.

4) For A_j, 1 (one) ≤ j ≤ m, then let $SDTA_j$ be the declared type of A_j.

5) Let $SDTP_{i,j}$ be the type designator of the declared type of the *j*-th SQL parameter of R_i.

6) For *r* varying from 1 (one) to *m*, if A_r is not a <dynamic parameter specification> and if there is more than one SQL-invoked routine in *SR*, then for each pair of SQL-invoked routines { R_p, R_q } in *SR*, if $SDTP_{p,r}$ $SDTP_{q,r}$ in the type precedence list of $SDTA_r$, then eliminate R_q from *SR*.

   NOTE 226 — The "type precedence list" of a given type is determined by Subclause 9.5, "Type precedence list determination".

7) The set of subject routines is the set of SQL-invoked routines remaining in *SR*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.5 Type precedence list determination

This Subclause is modified by Subclause 9.4, "Type precedence list determination", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.4, "Type precedence list determination", in ISO/IEC 9075-14.

Function

Determine the type precedence list of a given type.

Syntax Rules

1) Let *DT* be the data type specified in an application of this Subclause.

2) The *type precedence list TPL* of *DT* is a list of *type designators* as specified in the Syntax Rules of this Subclause.

3) Let "*A* *B*" represent "*A* has precedence over *B*" and let "*A* *B*" represent "*A* has the same precedence as *B*".

4) If *DT* is a user-defined type, then:

   a) Let *ST* be the set of supertypes of *DT*. Let *n* be the number of data types in *ST*.

   b) For any two data types *TA* and *TB* in *ST*, *TA* *TB* if and only if *TA* is a proper subtype of *TB*.

   c) Let T_1 be *DT* and let T_{i+1}, 1 (one) $\leq i \leq n-1$, be the direct supertype of T_i.

   d) Let DTN_i, 1 (one) $\leq i \leq n$, be the data type designator of T_i.

      NOTE 227 — The type designator of a user-defined type is the type name included in its user-defined type descriptor.

   e) *TPL* is DTN_1, DTN_2, ..., DTN_n.

5) If *DT* is fixed-length character string, then *TPL* is

```
CHARACTER, CHARACTER VARYING,
CHARACTER LARGE OBJECT
```

6) If *DT* is variable-length character string, then *TPL* is

```
CHARACTER VARYING, CHARACTER
LARGE OBJECT
```

7) If *DT* is fixed-length binary string, then *TPL* is

```
BINARY, BINARY VARYING, BINARY LARGE OBJECT
```

8) If *DT* is variable-length binary string, then *TPL* is

```
BINARY VARYING, BINARY LARGE OBJECT
```

9) If *DT* is numeric, then:

a) Let *NDT* be the following set of numeric types: NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, and DOUBLE PRECISION. For each type *T* in *NDT*, the *effective binary precision* is defined as follows.

Case:

i) If *T* is DECIMAL or NUMERIC, then the effective binary precision is the product of $\log_2(10)$ and the implementation-defined maximum precision of *T*.

ii) If *T* is FLOAT, then the effective binary precision is the implementation-defined maximum precision of *T*.

iii) If the radix of *T* is decimal, then the effective binary precision is the product of $\log_2(10)$ and the implementation-defined precision of *T*.

iv) Otherwise, the effective binary precision is the implementation-defined precision of *T*.

b) Let *PTC* be the set of all precedence relationships determined as follows: For any two types *T1* and *T2*, not necessarily distinct, in *NDT*,

Case:

i) If *T1* is exact numeric and *T2* is approximate numeric, then *T1* *T2*.

ii) If *T1* is approximate numeric and *T2* is exact numeric, then *T1* *T2*.

iii) If the effective binary precision of *T1* is greater than the effective binary precision of *T2*, then *T2* *T1*.

iv) If the effective binary precision of *T1* equals the effective binary precision of *T2*, then *T2* *T1*.

v) Otherwise, *T1* *T2*.

c) *TPL* is determined as follows:

i) *TPL* is initially empty.

ii) Let *ST* be the set of types containing *DT* and every type *T* in *NDT* for which the precedence relationship *DT* *T* or *DT* *T* is in *PTC*.

iii) Let *n* be the number of types in *ST*.

iv) For *i* ranging from 1 (one) to *n*:

1) Let *NT* be the set of types T_k in *ST* such that there is no other type T_j in *ST* for which T_j T_k according to *PTC*.

2) Case:

A) If there is exactly one type T_k in *NT*, then T_k is placed next in *TPL* and all relationships of the form T_k T_r are removed from *PTC*, where T_r is any type in *ST*.

B) If there is more than one type T_k in *NT*, then every type T_s in *NT* is assigned the same position in *TPL* as T_k and all relationships of the forms T_k T_r, T_k T_r, T_s T_r, and T_s T_r are removed from *PTC*, where T_r is any type in *ST*.

10) If *DT* specifies a year-month interval type, then *TPL* is

```
INTERVAL YEAR
```

11) If *DT* specifies a day-time interval type, then *TPL* is

```
INTERVAL DAY
```

12) If *DT* specifies DATE, then *TPL* is

```
DATE
```

13) If *DT* specifies TIME, then *TPL* is

```
TIME
```

14) If *DT* specifies TIMESTAMP, then *TPL* is

```
TIMESTAMP
```

15) If *DT* specifies BOOLEAN, then *TPL* is

```
BOOLEAN
```

16) If *DT* is a collection type, then let *CTC* be the kind of collection (either ARRAY or MULTISET) specified in *DT*.

Let *n* be the number of elements in the type precedence list for the element type of *DT*. For *i* ranging from 1 (one) to *n*, let RIO_i be the *i*-th such element. *TPL* is

RIO_1 *CTC*,
RIO_2 *CTC*, ...,
RIO_n *CTC*

17) If *DT* is a reference type, then let *n* be the number of elements in the type precedence list for the referenced type of *DT*. For *i* ranging from 1 (one) to *n*, let KAW_i be the *i*-th such element. *TPL* is

REF(KAW_1),
REF(KAW_2), ...,
REF(KAW_n)

18) If *DT* is a row type, then *TPL* is

```
ROW
```

NOTE 228 — This rule is placed only to avoid the confusion that might arise if row types were not mentioned in this Subclause. As a row type cannot be used as a <parameter type>, the type precedence list of a row type is never referenced.

Conformance Rules

None.

9.6 Host parameter mode determination

Function

Determine the parameter mode for a given host parameter.

Syntax Rules

1) Let *PD* and *SPS* be a <host parameter declaration> and an <SQL procedure statement> specified in an application of this Subclause.

2) Let *P* be the host parameter specified by *PD* and let *PN* be the <host parameter name> immediately contained in *PD*.

3) Whether *P* is an input host parameter, an output host parameter, or both an input host parameter and an output host parameter is determined as follows.

   Case:

   a) If *PD* is a <status parameter>, then *P* is an output host parameter.

   b) Otherwise,

      Case:

      i) If *PN* is contained in an <SQL argument> A_i of the <SQL argument list> of a <routine invocation> immediately contained in a <call statement> that is contained in *SPS*, then:

         1) Let *R* be the subject routine of the <routine invocation>.

         2) Let PR_i be the *i*-th SQL parameter of *R*.

         3) Case:

            A) If *PN* is contained in a <host parameter specification> that is the <target specification> that is simply contained in A_i and PR_i is an output SQL parameter, then *P* is an output host parameter.

            B) If *PN* is contained in a <host parameter specification> that is the <target specification> that is simply contained in A_i and PR_i is both an input SQL parameter and an output SQL parameter, then *P* is both an input host parameter and an output host parameter.

            C) Otherwise, *P* is an input host parameter.

      ii) If *PN* is contained in a <value specification> or a <simple value specification> that is contained in *SPS*, and *PN* is not contained in a <target specification> or a <simple target specification> that is contained in *SPS*, then *P* is an input host parameter.

      iii) If *PN* is contained in a <target specification> or a <simple target specification> that is contained in *SPS*, and *PN* is not contained in a <value specification> or a <simple value specification> that is contained in *SPS*, then *P* is an output host parameter.

iv) If *PN* is contained in a <value specification> or a <simple value specification> that is contained in *SPS*, and in a <target specification> or a <simple target specification> that is contained in *SPS*, then *P* is both an input host parameter and an output host parameter.

v) Otherwise, *P* is neither an input host parameter nor an output host parameter.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.7 Type name determination

This Subclause is modified by Subclause 10.5, "Type name determination", in ISO/IEC 9075-14.

Function

Determine an <identifier> given the name of a predefined data type.

Syntax Rules

1) Let *DT* be the <predefined type> specified in an application of this Subclause.
2) Let *FNSDT* be the <identifier> resulting from an application of this Subclause, defined as follows.

   Case:

   a) If *DT* specifies CHARACTER, then let *FNSDT* be "CHAR".
   b) If *DT* specifies CHARACTER VARYING, then let *FNSDT* be "VARCHAR".
   c) If *DT* specifies CHARACTER LARGE OBJECT, then let *FNSDT* be "CLOB".
   d) If *DT* specifies BINARY, then let *FNSDT* be "BINARY".
   e) If *DT* specifies BINARY VARYING, then let *FNSDT* be "VARBINARY".
   f) If *DT* specifies BINARY LARGE OBJECT, then let *FNSDT* be "BLOB".
   g) If *DT* specifies SMALLINT, then let *FNSDT* be "SMALLINT".
   h) If *DT* specifies INTEGER, then let *FNSDT* be "INTEGER".
   i) If *DT* specifies BIGINT, then let *FNSDT* be "BIGINT".
   j) If *DT* specifies DECIMAL, then let *FNSDT* be "DECIMAL".
   k) If *DT* specifies NUMERIC, then let *FNSDT* be "NUMERIC".
   l) If *DT* specifies REAL, then let *FNSDT* be "REAL".
   m) If *DT* specifies FLOAT, then let *FNSDT* be "FLOAT".
   n) If *DT* specifies DOUBLE PRECISION, then let *FNSDT* be "DOUBLE".
   o) If *DT* specifies DATE, then let *FNSDT* be "DATE".
   p) If *DT* specifies TIME, then let *FNSDT* be "TIME".
   q) If *DT* specifies TIMESTAMP, then let *FNSDT* be "TIMESTAMP".
   r) If *DT* specifies INTERVAL, then let *FNSDT* be "INTERVAL".

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.8 Determination of identical values

This Subclause is modified by Subclause 9.5, "Determination of identical values", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.6, "Determination of identical values", in ISO/IEC 9075-14.

Function

Determine whether two instances of values are identical, that is to say, are occurrences of the same value.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *V1* and *V2* be two values specified in an application of this Subclause.

   NOTE 229 — This Subclause is invoked implicitly wherever the word *identical* is used of two values.

2) Case:

   a) If *V1* and *V2* are both the null value, then *V1* is identical to *V2*.

   b) If *V1* is the null value and *V2* is not the null value, or if *V1* is not the null value and *V2* is the null value, then *V1* is not identical to *V2*.

   c) If *V1* and *V2* are of comparable predefined types, then

      Case:

      i) If *V1* and *V2* are character strings, then let *L* be CHARACTER_LENGTH(*V1*).

         Case:

         1) If CHARACTER_LENGTH(*V2*) equals *L*, and if for all *i*, 1 (one) ≤ *i* ≤ *L*, the *i*-th character of *V1* corresponds to the same character position of ISO/IEC 10646 as the *i*-th character of *V2*, then *V1* is identical to *V2*.

         2) Otherwise, *V1* is not identical to *V2*.

      ii) If *V1* and *V2* are TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE and are not distinct, and their time zone displacement fields are not distinct, then *V1* is identical to *V2*.

      iii) Otherwise, *V1* is identical to *V2* if and only if *V1* is not distinct from *V2*.

   d) If *V1* and *V2* are of constructed types, then

      Case:

i) If *V1* and *V2* are rows and their respective fields are identical, then *V1* is identical to *V2*.

ii) If *V1* and *V2* are arrays and have the same cardinality and elements in the same ordinal position in the two arrays are identical, then *V1* is identical to *V2*.

iii) If *V1* and *V2* are multisets and have the same cardinality *N* and there exist enumerations $VE1_i$, 1 (one) ≤ i ≤ N, of *V1* and $VE2_i$, 1 (one) ≤ i ≤ N, of *V2* such that for all i, $VE1_i$ is identical to $VE2_i$, then *V1* is identical to *V2*.

iv) If *V1* and *V2* are references and *V1* is not distinct from *V2*, then *V1* is identical to *V2*.

v) Otherwise, *V1* is not identical to *V2*.

e) If *V1* and *V2* are of the same most specific type *MST* and *MST* is a user-defined type, then

Case:

i) If *MST* is a distinct type whose source type is *SDT* and the results of `SDT(V1)` and `SDT(V2)` are identical, then *V1* is identical to *V2*.

ii) If *MST* is a structured type and, for every observer function *O* defined for *MST*, the results of the invocations *O*(*V1*) and *O*(*V2*) are identical, then *V1* is identical to *V2*.

iii) Otherwise, *V1* is not identical to *V2*.

f) Otherwise, *V1* is not identical to *V2*.

Conformance Rules

None.

9.9 Equality operations

This Subclause is modified by Subclause 9.6, "Equality operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.8, "Equality operations", in ISO/IEC 9075-14.

Function

Specify the prohibitions and restrictions by data type on operations that involve testing for equality.

Syntax Rules

1) An *equality operation* is any of the following:

   a) A <comparison predicate> that specifies <equals operator> or <not equals operator>.

   b) A <quantified comparison predicate> that specifies <equals operator> or <not equals operator>.

   c) An <in predicate>.

   d) A <like predicate>.

   e) A <similar predicate>.

   f) A <distinct predicate>.

   g) A <match predicate>.

   h) A <member predicate>.

   i) A <joined table> that specifies NATURAL or USING.

   j) A <user-defined ordering definition> that specifies MAP.

   k) A <position expression>.

2) An *operand of an equality operation* is any of the following:

   a) A field of the declared row type of a <row value predicand> that is simply contained in a <comparison predicate> that specifies <equals operator> or <not equals operator>.

   b) A field of the declared row type of a <row value predicand> that is simply contained in a <quantified comparison predicate> that specifies <equals operator> or <not equals operator>.

   c) A column of a <table subquery> that is simply contained in a <quantified comparison predicate> that specifies <equals operator> or <not equals operator>.

   d) A field of the declared row type of a <row value predicand> or <row value expression> that is simply contained in an <in predicate>.

   e) A column of a <table subquery> that is simply contained in an <in predicate>.

   f) A field of the declared row type of a <row value predicand> that is simply contained in a <like predicate>.

g) A <character pattern>, <escape character>, <octet pattern> or <escape octet> that is simply contained in a <like predicate>.

h) A field of the declared row type of a <row value predicand> that is simply contained in a <similar predicate>.

i) A <similar pattern> or <escape character> that is simply contained in a <similar predicate>.

j) A field of the declared row type of a <row value predicand> that is simply contained in a <distinct predicate>.

k) A field of the declared row type of a <row value predicand> that is simply contained in a <match predicate>.

l) A column of a <table subquery> that is simply contained in a <match predicate>.

m) A field of the declared row type of a <row value predicand> that is simply contained in a <member predicate>.

n) A corresponding join column of a <joined table> that specifies NATURAL or USING.

o) The <returns data type> of the SQL-invoked function identified by a <map function specification> simply contained in a <user-defined ordering definition> that specifies MAP.

p) A <string value expression> that is simply contained in a <position expression>.

q) A <binary value expression> that is simply contained in a <position expression>.

3) The declared type of an operand of an equality operation shall not be UDT-NC-ordered.

4) If the declared type of the operands of an equality operation is a character string type, then the Syntax Rules of Subclause 9.13, “Collation determination”, apply.

5) If the declared type of an operand *OP* of an equality operation is a multiset type, then *OP* is a multiset operand of a multiset element grouping operation. The Syntax Rules of Subclause 9.11, “Multiset element grouping operations”, apply.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, “Enhanced structured types”, in conforming SQL language, the declared type of an operand of an equality operation shall not be ST-ordered.

2) Without Feature T042, “Extended LOB data type support”, in conforming SQL language, the declared type of an operand of an equality operation shall not be LOB-ordered.

3) Without Feature S275, “Advanced multiset support”, in conforming SQL language, the declared type of an operand of an equality operation shall not be multiset-ordered.

NOTE 230 — If the declared type of an operand *OP* of an equality operation is a multiset type, then *OP* is a multiset operand of a multiset element grouping operation. The Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, apply.

9.10 Grouping operations

This Subclause is modified by Subclause 9.7, "Grouping operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.9, "Grouping operations", in ISO/IEC 9075-14.

Function

Specify the prohibitions and restrictions by data type on operations that involve grouping of data.

Syntax Rules

1) A *grouping operation* is any of the following:
   a) A <group by clause>.
   b) A <window partition clause>.
   c) An <aggregate function> that specifies DISTINCT.
   d) A <query specification> that immediately contains DISTINCT.
   e) A <query expression body> that simply contains or implies UNION DISTINCT.
   f) A <query expression body> that simply contains EXCEPT.
   g) A <query term> that simply contains INTERSECT.
   h) A <unique predicate>.
   i) A <unique constraint definition>.
   j) A <referential constraint definition>.
2) An *operand of a grouping operation* is any of the following:
   a) A grouping column of a <group by clause>.
   b) A partitioning column of a <window partition clause>.
   c) A <value expression> simply contained in an <aggregate function> that specifies DISTINCT.
   d) A column of the result of a <query specification> that immediately contains DISTINCT.
   e) A column of the result of a <query expression body> that simply contains or implies UNION DISTINCT.
   f) A column of the result of a <query expression body> that simply contains EXCEPT.
   g) A column of the result of a <query term> that simply contains INTERSECT.
   h) A column of the <table subquery> simply contained in a <unique predicate>.
   i) A column identified by the <unique column list> of a <unique constraint definition>.
   j) A referencing column of a <referential constraint definition>.

3) The declared type of an operand of a grouping operation shall not be LOB-ordered, array-ordered, multiset-ordered, UDT-EC-ordered, or UDT-NC-ordered.

4) If the declared type of an operand of a grouping operation is a character string type, then the Syntax Rules of Subclause 9.13, "Collation determination", apply.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", in conforming SQL language, the declared type of an operand of a grouping operation shall not be ST-ordered.

9.11 Multiset element grouping operations

This Subclause is modified by Subclause 9.8, "Multiset element grouping operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.10, "Multiset element grouping operations", in ISO/IEC 9075-14.

Function

Specify the prohibitions and restrictions by data type on the declared element type of a multiset for operations that involve grouping the elements of a multiset.

Syntax Rules

1) A *multiset element grouping operation* is any of the following:

   a) An equality operation such that the declared type of an operand of the equality operation is a multiset type.

   b) A <multiset set function>.

   c) A <multiset value expression> that specifies MULTISET UNION DISTINCT.

   d) A <multiset value expression> that specifies MULTISET EXCEPT.

   e) A <multiset term> that specifies MULTISET INTERSECT.

   f) A <submultiset predicate>.

   g) A <set predicate>.

   h) A <general set function> that specifies INTERSECTION.

2) A *multiset operand* of a multiset element grouping operation is any of the following:

   a) A <multiset value expression> simply contained in a <multiset set function>.

   b) A <multiset value expression> or a <multiset term> that is simply contained in a <multiset value expression> that simply contains MULTISET UNION DISTINCT.

   c) A <multiset value expression> or a <multiset term> that is simply contained in a <multiset value expression> that simply contains MULTISET EXCEPT.

   d) A <multiset term> or a <multiset primary> that is simply contained in a <multiset term> that simply contains MULTISET INTERSECT.

   e) A field of a comparand of a <comparison predicate> such that the <comparison predicate> specifies <equals operator> or <not equals operator> and such that the declared type of the field is a multiset type.

   f) A <multiset value expression> simply contained in a <member predicate>.

   g) A field of the <row value expression> simply contained in a <submultiset predicate>.

   h) The <multiset value expression> simply contained in a <submultiset predicate>.

   i) A field of the <row value expression> simply contained in a <set predicate>.

j) A value expression> simply contained in a <general set function> that specifies INTERSECTION.

3) The declared element type of a multiset operand of a multiset element grouping operation shall not be LOB-ordered, array-ordered, multiset-ordered, UDT-EC-ordered, or UDT-NC-ordered.

4) If the declared element type of a multiset operand of a multiset element grouping operation is a character string type, then the Syntax Rules of Subclause 9.13, "Collation determination", apply.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", in conforming SQL language, the declared element type of a multiset operand of a multiset element grouping operation shall not be ST-ordered.

9.12 Ordering operations

This Subclause is modified by Subclause 9.9, "Ordering operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.11, "Ordering operations", in ISO/IEC 9075-14.

Function

Specify the prohibitions and restrictions by data type on operations that involve ordering of data.

Syntax Rules

1) An *ordering operation* is any of the following:

   a) A <comparison predicate> that does not specify <equals operator> or <not equals operator>.

   b) A <quantified comparison predicate> that does not specify <equals operator> or <not equals operator>.

   c) A <between predicate>.

   d) An <overlaps predicate>.

   e) An <aggregate function> that specifies MAX or MIN.

   f) A <sort specification list>.

   g) A <user-defined ordering definition> that specifies ORDER FULL BY MAP.

2) An *operand of an ordering operation* is any of the following:

   a) A field of the declared row type of a <row value predicand> that is simply contained in a <comparison predicate> that does not specify <equals operator> or <not equals operator>.

   b) A field of the declared row type of a <row value predicand> that is simply contained in a <quantified comparison predicate> that does not specify <equals operator> or <not equals operator>.

   c) A column of the <table subquery> that is simply contained in a <quantified comparison predicate> that does not specify <equals operator> or <not equals operator>.

   d) A field of the declared row type of a <row value predicand> that is simply contained in a <between predicate>.

   e) A field of the declared row type of a <row value predicand> that is simply contained in an <overlaps predicate>.

   f) A <value expression> simply contained in an <aggregate function> that specifies MAX or MIN.

   g) A <value expression> simply contained in a <sort key>.

   h) The <returns data type> of the SQL-invoked function identified by a <map function specification> simply contained in a <user-defined ordering definition> that specifies ORDER FULL BY MAP.

3) The declared type of an operand of an ordering operation shall not be LOB-ordered, array-ordered, multiset-ordered, reference-ordered, UDT-EC-ordered, or UDT-NC-ordered.

4) If the declared type of an operand of an ordering operation is a character string type, then the Syntax Rules of Subclause 9.13, "Collation determination", apply.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", in conforming SQL language, the declared type of an operand of an ordering operation shall not be ST-ordered.

9.13 Collation determination

Function

Specify rules for determining the collation to be used in the comparison of character strings.

Syntax Rules

1) Let *CCS* be the character set of the result of applying the rules of Subclause 9.3, "Result of data type combinations", to the declared types of all operands of the comparison operation.

2) At least one operand shall have a declared type collation.

3) Case:

   a) If the comparison operation is a <referential constraint definition>, then, for each referencing column, the collation to be used is the declared type collation of the corresponding column of the referenced table.

   b) If at least one operand has an *explicit* collation derivation, then every operand whose collation derivation is *explicit* shall have the same declared type collation *EDTC* and the collation to be used is *EDTC*.

   c) If the comparison operation is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement>, or in a <direct SQL statement> that is invoked directly, and *CCS* has an SQL-session collation, then the collation to be used is that SQL-session collation.

   d) If *CCS* has an SQL-client module collation, then the collation to be used is that collation.

   e) Otherwise, every operand whose collation derivation is *implicit* shall have the same declared type collation *IDTC* and the collation to be used is *IDTC*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.14 Execution of array-returning functions

This Subclause is modified by Subclause 8.5, "Execution of array-returning functions", in ISO/IEC 9075-13.

Function

Define the execution of an external function that returns an array value.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *AR*, *ESPL*, and *P* be the *ARRAY*, *EFFECTIVE SQL PARAMETER LIST*, and *PROGRAM* specified in an application of this Subclause.

2) Let *ARC* be the cardinality of *AR*.

3) Let *EN* be the number of entries in *ESPL*.

4) Let ESP_i, 1 (one) ≤ i ≤ *EN*, be the i-th parameter in *ESPL*.

5) Let *FRN* be the number of result data items.

6) Let *PN* and *N* be the number of values in the static SQL argument list of *P*.

7) Let *E* be 0 (zero).

8) If the call type data item has a value of –1 (indicating "open call"), then *P* is executed with a list of *EN* parameters PD_i whose parameter names are PN_i and whose values are set as follows:

   a) Depending on whether the language of *R* specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the *operative data type correspondences* table be Table 16, "Data type correspondences for Ada", Table 17, "Data type correspondences for C", Table 18, "Data type correspondences for COBOL", Table 19, "Data type correspondences for Fortran", Table 20, "Data type correspondences for M", Table 21, "Data type correspondences for Pascal", or Table 22, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the "SQL data type" column and the "host data type" column.

   b) For i varying from 1 (one) to *EN*, the data type DT_i of PD_i is the data type listed in the host data type column of the row in the data type correspondences table whose value in the SQL data type column corresponds to the data type of ESP_i.

   c) The value of PD_i is set to the value of ESP_i.

9) Case:

   a) If the value of the exception data item is '00000' (corresponding to the completion condition *successful completion*) or the first 2 characters are '01' (corresponding to the completion condition *warning* with any subcondition), then set the call type data item to 0 (zero) (indicating *fetch call*).

   b) If the exception data item is '02000' (corresponding to the completion condition *no data*):

      i) If each PD_i, for i ranging from (PN+FRN)+N+1 through (PN+FRN)+N+FRN (that is, the SQL indicator arguments corresponding to the result data items), has the value –1, then set *AR* to the null value.

      ii) Set the call type data item to 1 (one) (indicating *close call*).

   c) Otherwise, set the call type data item to 1 (one) (indicating *close call*).

10) The following steps are applied as long as the call type data item has a value 0 (zero) (corresponding to *fetch call*):

   a) *P* is executed with a list of *EN* parameters PD_i whose parameter names are PN_i and whose values are set as follows:

      i) For i varying from 1 (one) to *EN*, the <data type> DT_i of PD_i is the data type listed in the host data type column of the row in the data type correspondences table whose value in the SQL data type column corresponds to the data type of ESP_i.

      ii) For i ranging from 1 (one) to *EN*–2, the value of PD_i is set to the value of ESP_i.

      iii) For the save area data item, for i equal to *EN*–1, the value of PD_i is set to the value returned in PD_i by the prior execution of *P*.

      iv) For the call type data item, for i equal to *EN*, the value of PD_i is set to 0 (zero).

   b) Case:

      i) If the exception data item is '00000' (corresponding to completion condition *successful completion*) or the first 2 characters are '01' (corresponding to completion condition *warning* with any subcondition), then:

         1) Increment *E* by 1 (one).

         2) If *E* > *ARC*, then an exception condition is raised: *data exception — array element error*.

         3) If the call type data item is 0 (zero), then

            Case:

            A) If each PD_i, for i ranging from (PN+FRN)+N+1 through (PN+FRN)+N+FRN (that is, the SQL indicator arguments corresponding to the result data items) is negative, then let the *E*-th element of *AR* be the null value.

            B) Otherwise,

               Case:

               I) If *FRN* is 1 (one), then let the *E*-th element of *AR* be the value of the result data item.

II) Otherwise:

1) Let RDI_i, 1 (one) ≤ i ≤ FRN, be the value of the i-th result data item.

2) Let the E-th element of AR be the value of the following <row value expression>:

```
ROW ( RDI1, ... ,
RDIFRN )
```

ii) If the exception data item is '02000' (corresponding to completion condition *no data*), then:

1) If the value of E is 0 (zero), then set AR to an empty array.

2) Set the call type data item to 1 (one) (indicating *close call*).

iii) Otherwise, set the value of the call type data item to 1 (one) (indicating *close call*).

11) If the call type data item has a value of 1 (one) (indicating *close call*), then P is executed with a list of EN parameters PD_i whose parameter names are PN_i and whose values are set as follows:

a) For i varying from 1 (one) to EN, the <data type> DT_i of PD_i is the data type listed in the host data type column of the row in the data type correspondences table whose value in the SQL data type column corresponds to the data type of ESP_i.

b) For i ranging from 1 (one) to EN–2, the value of PD_i is set to the value of ESP_i.

c) For the save area data item, for i equal to EN–1, the value of PD_i is set to the value returned in PD_i by the prior execution of P.

d) For the call type data item, for i equal to EN, the value of PD_i is set to 1 (one).

Conformance Rules

None.

9.15 Execution of multiset-returning functions

Function

Define the execution of an external function that returns a multiset value.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *MU*, *ESPL*, and *P* be the *MULTISET*, *EFFECTIVE SQL PARAMETER LIST*, and *PROGRAM* specified in an application of this subclause.

2) Let *ET* be the element type of *MU*.

3) Let *C* be the maximum implementation-defined cardinality of array type with element type *ET*.

4) Let *AT* be the array type *ET* ARRAY[*C*].

5) Let *AR* be an array whose declared type is *AT*.

6) The General Rules of Subclause 9.14, "Execution of array-returning functions", are applied with *AR*, *ESPL*, and *P* as *ARRAY*, *EFFECTIVE SQL PARAMETER LIST*, and *PROGRAM*, respectively.

7) Let *MU* be the result of casting *AR* to the multiset type of *MU* according to the General Rules of Subclause 6.12, "<cast specification>".

Conformance Rules

None.

9.16 XQuery regular expression matching

Function

Determine a list of match vectors to an XQuery regular expression in a character string.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the *STRING*, *PAT* be the *PATTERN*, *SP* be the *POSITION*, *U* be the *UNITS*, and *FL* be the *FLAG* in an application of this Subclause.

2) Case:

   a) If the character repertoire of *PAT* is UCS, then let *P* be *PAT*.

   b) Otherwise, let *P* be the result of an implementation-defined conversion of *PAT* to UCS.

3) Case:

   a) If the character repertoire of *FL* is UCS, then let *F* be *FL*.

   b) Otherwise, let *F* be the result of an implementation-defined conversion of *FL* to UCS.

4) If *P* is not an XQuery regular expression, then an exception condition is raised: *data exception — invalid XQuery regular expression*.

5) If *F* is not an XQuery option flag, then an exception condition is raised: *data exception — invalid XQuery option flag*.

6) Let *LEN* be the length in characters of *S*.

7) A position/length *PL* is a pair of exact numeric values, a position *P* and a length *L*, such that either $P = 0$ (zero) and $L = 0$ (zero), or *P* is positive, *L* is non-negative, and $P + L \leq LEN + 1$ (one). If *P* is positive, then *PL* represents the substring of *S* whose starting position is *P* and whose length is *L*. If *P* is 0 (zero), then *PL* does not represent a substring of *S*.

   NOTE 231 — For example, if *S* is `'a'`, then there are four position/lengths, namely (1,0), denoting the zero-length substring at the beginning of *S*; (2,0), denoting he zero-length substring at the end of *S*; (1,1), denoting the whole string *S*; and (0,0), denoting no substring of *S*.

8) Let *G* be the number of parenthesized groups in *PAT*.

9) An *XQuery regular expression match* is defined as follows:

   Case:

a) If the character repertoire of *S* is UCS, then an XQuery regular expression match is a match as defined by [XQueryFO] for the function `fn:matches()`, with *F* as the value of the parameter `$flags`, with the following modifications:

   i) The metacharacters "`^`", "`$`", and "`.`" are interpreted according to the guidelines of RL1.6 "Line boundaries", in [Unicode18].

   ii) The multi-character escape "`\s`" matches U+0020 (space), U+0009 (tab), or any single character or two-character sequence identified as a newline sequence by RL1.6 "line boundaries" in [Unicode18].

b) Otherwise, the definition of XQuery regular expression match is implementation-defined.

10) A match vector *MV* is an array of *G* + 1 position/lengths, indexed from 0 (zero) through *G*. *MV* represents an XQuery regular expression match of *PAT*, including information about capture groups. *MV*[0] is the position/length that denotes the entire XQuery regular expression match of *PAT*. For *i* between 1 (one) and *G*, *MV*[*i*] is the position/length that denotes the *i*-th capture group within the XQuery regular expression match that is denoted by *MV*[0]. If the *i*-th parenthesized subexpression of *PAT* is not matched, then the position of *MV*[*i*] is 0 (zero).

NOTE 232 — The position of *MV*[0] is never 0 (zero), since an XQuery regular expression match to *PAT* can never be "no substring". An example of a capture group that is not matched occurs when *S* is `'a'` and *PAT* is `'(a)|(b)'`. In this example, the match vector is given by *MV*[0] = (1,1), *MV*[1] = (1,1), and *MV*[2] = (0,0), since the second parenthesized subexpression has no match.

11) The result of this Subclause is a list *LIST* of match vectors, defined as follows:

a) Let P_0 be *SP* – 1 (one), and let L_0 be 0 (zero).

b) For $i \geq 1$ (one), P_i and L_i are defined recursively as follows: Let P_i be the position in units *U* of the first XQuery regular expression match of *PAT* in *S* such that $P_i > P_{i-1} + L_{i-1}$, and let L_i be the length in units *U* of this match.

NOTE 233 — The sequence P_i is strictly increasing and bounded above by *LEN* + 1 (one), so this recursive definition terminates after finding a finite number of disjoint matches.

c) Let *N* be the number of XQuery regular expression matches found.

NOTE 234 — It is possible that there are no matches, in which case *N* is 0 (zero), and *LIST* is empty. (P_0, L_0) = (*SP* – 1, 0) is not included in the result; it is merely used as the position to start seeking the first match.

d) For *i* between 1 (one) and *N*, let MV_i be the match vector such that:

   i) MV_i[0] is the position/length whose position is P_i and whose length is L_i.

   ii) For *j* between 1 (one) and *G*, MV_i[*j*] is the position/length

   Case:

      1) If the *j*-th parenthesized sub-expression of *PAT* is not matched in the *i*-th XQuery regular expression match, then position 0 (zero) and length 0 (zero).

      2) Otherwise, the position and length of the *j*-th capture group within the *i*-th XQuery regular expression match.

e) Let *LIST* be the list of *N* match vectors MV_i in order from 1 (one) to *N*.

12) *LIST* is the result of this Subclause.

Conformance Rules

None.

9.17 XQuery regular expression replacement

Function

Replace a substring that matches an XQuery regular expression with a replacement character string.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *STR* be the *STRING*, *PAT* be the *PATTERN*, *MV* be the *MATCH*, *REP* be the *REPLACEMENT*, and *FL* be the *FLAG* in an application of this Subclause.

2) Let *PL* be *MV*[0]. Let *P* be the position of *PL* and let *L* be the length of *PL*.

3) If *L* is zero, then an exception condition is raised: *data exception — attempt to replace a zero-length string*.

4) Case:

   a) If the character repertoire of *REP* is UCS, then let *R* be *REP*.

   b) Otherwise, let *R* be the result of an implementation-defined conversion of *REP* to UCS.

5) If *R* is not an XQuery replacement string, then an exception condition is raised: *data exception — invalid XQuery replacement string*.

6) Let STR_1 be the character string consisting of the first *P* characters of *STR*.

7) Let STR_2 be the character string consisting of the *P*-th through (*P*+*L*—1)-th character of *STR*.

8) Let STR_3 be the character string consisting of the (*P*+*L*—1)-th character through the last character of *STR*.

9) Case:

   a) If the character repertoire of *STR*, *PAT*, *REP*, and *FL* is UCS, then let *T* be the result of the XQuery function `fn:replace()`, where the value of the `$input` argument is STR_2, the value of the `$pattern` argument is *P*, the value of the `$replacement` argument is *R*, and the value of the `$flags` argument is *F*.

   b) Otherwise, the value of *T* is implementation-defined.

10) The result of this Subclause is the concatenation of STR_1, *T*, and STR_3.

Conformance Rules

None.

9.18 Data type identity

Function

Determine whether two data types are compatible and have the same characteristics.

Syntax Rules

1) Let *PM* and *P* be the two data types specified in an application of this Subclause.

2) *PM* and *P* shall be compatible.

3) If *PM* is a fixed-length character string type, then the length of *PM* shall be equal to the length of *P*.

4) If *PM* is a variable-length character string type, then the maximum length of *PM* shall be equal to the maximum length of *P*.

5) If *PM* is an exact numeric type, then the precision and scale of *PM* shall be equal to the precision and scale of *P*, respectively.

6) If *PM* is an approximate numeric type, then the precision of *PM* shall be equal to the precision of *P*.

7) If *PM* is a fixed-length binary string type, then the length of *PM* shall be equal to the length of *P*.

8) If *PM* is a variable-length binary string type, then the maximum length of *PM* shall be equal to the maximum length of *P*.

9) If *PM* is a datetime data type with <time fractional seconds precision>, then the <time fractional seconds precision> of *PM* shall be equal to the <time fractional seconds precision> of *P*.

10) If *PM* is an interval type, then the <interval qualifier> of *PM* shall be equivalent to the <interval qualifier> of *P*.

11) If *PM* is a collection type, then:

    a) The kind of collection (ARRAY or MULTISET) of *PM* and the kind of collection of *P* shall be the same.

    b) If *PM* is an array type, then the maximum cardinality of *PM* shall be equal to the the maximum cardinality of *P*.

    c) The Syntax Rules of this Subclause are applied with the element type of *PM* and the element type of *P* as the two data types.

12) If *PM* is a row type, then:

    a) Let *N* be the degree of *PM*.

    b) Let $DTFPM_i$ and $DTFP_i$, 1 (one) $\leq i \leq N$, be the data type of the *i*-th field of *PM* and of *P*, respectively. For *i* varying from 1 (one) to *N*, the Syntax Rules of this Subclause are applied with $DTFPM_i$ and $DTFP_i$ the two data types.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.19 Determination of a from-sql function

Function

Determine the from-sql function of a user-defined type given the name of a user-defined type and the name of the group.

Syntax Rules

1) Let *UDT* and *GN* be a *TYPE* and a *GROUP* specified in an application of this Subclause.

2) Let *SSUDT* be the set of supertypes of *UDT*.

3) Let *SUDT* be the data type, if any, in *SSUDT* such that the transform descriptor included in the data type descriptor of *SUDT* includes a group descriptor *GD* that includes a group name that is equivalent to *GN*.

4) The *applicable from-sql function* is the SQL-invoked function identified by the specific name of the from-sql function, if any, in *GD*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.20 Determination of a from-sql function for an overriding method

Function

Determine the from-sql function of a user-defined type given the name of an overriding method and the ordinal position of an SQL parameter.

Syntax Rules

1) Let *R* and *N* be a *ROUTINE* and a *POSITION* specified in an application of this Subclause.

2) Let *OM* be original method of *R*.

3) The *applicable from-sql function* is the from-sql function associated with the *N*-th SQL parameter of *OM*, if any.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.21 Determination of a to-sql function

Function

Determine the to-sql function of a user-defined type given the name of a user-defined type and the name of a group.

Syntax Rules

1) Let *UDT* and *GN* be a *TYPE* and a *GROUP* specified in an application of this Subclause.

2) Let *SSUDT* be the set of supertypes of *UDT*.

3) Let *SUDT* be the data type, if any, in *SSUDT* such that the transform descriptor included in the data type descriptor of *SUDT* includes a group descriptor *GD* that includes a group name that is equivalent to *GN*.

4) The *applicable to-sql function* is the SQL-invoked function identified by the specific name of the to-sql function, if any, in *GD*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.22 Determination of a to-sql function for an overriding method

Function

Determine the to-sql function of a user-defined type given the name of an overriding method.

Syntax Rules

1) Let *R* be a *ROUTINE* specified in an application of this Subclause.

2) Let *OM* be the original method of *R*

3) The *applicable to-sql function* is the SQL-invoked function associated with the result of *OM*, if any.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.23 Generation of the next value of a sequence generator

Function

Generate and return the next value of a sequence generator.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *SEQ* be the *SEQUENCE* specified in an application of this Subclause.

2) Let *DT*, *CBV*, *INC*, *SMAX*, and *SMIN* be the data type, current base value, increment, maximum value and minimum value, respectively, of *SEQ*.

3) If there exists a non-negative integer *N* such that $SMIN \leq CBV + N * INC \leq SMAX$ and the value ($CBV + N * INC$) has not already been returned in the current cycle, then let *V1* be ($CBV + N * INC$). Otherwise,

   Case:

   a) If the cycle option of *SEQ* is NO CYCLE, then an exception condition is raised: *data exception — sequence generator limit exceeded*.

   b) Otherwise, a new cycle is initiated.

      Case:

      i) If *SEQ* is an ascending sequence generator, then let *V1* be *SMIN*.

      ii) Otherwise, let *V1* be *SMAX*.

4) Case:

   a) If *SEQ* is an ascending sequence generator, the current base value of *SEQ* is set to the value of the lowest non-issued value in the cycle.

   b) Otherwise, the current base value of *SEQ* is set to the highest non-issued value in the cycle.

5) *V1* is returned as the *RESULT*.

Conformance Rules

None.

9.24 Creation of a sequence generator

Function

Complete the definition of an external or internal sequence generator.

Syntax Rules

1) Let *OPT* and *DT* be the *OPTIONS* and *DATA TYPE* specified in an application of this Subclause. *OPT* shall conform to the Format of <common sequence generator options>. The BNF nonterminal symbols used in the remainder of this Subclause refer to the contents of *OPT*.

2) Each of <sequence generator start with option>, <sequence generator increment by option>, <sequence generator maxvalue option>, <sequence generator minvalue option>, and <sequence generator cycle option> shall be specified at most once.

3) If <sequence generator increment by option> is specified, then let *INC* be <sequence generator increment>; otherwise, let *INC* be a <signed numeric literal> whose value is 1 (one).

4) The value of *INC* shall not be 0 (zero).

5) If the value of *INC* is negative, then *SEQ* is a *descending sequence generator*; otherwise, *SEQ* is an *ascending sequence generator*.

6) Case:

   a) If <sequence generator maxvalue option> is specified, then

      Case:

      i) If NO MAXVALUE is specified, then let *SMAX* be an implementation-defined <signed numeric literal> of declared type *DT*.

      ii) Otherwise, let *SMAX* be <sequence generator max value>.

   b) Otherwise, let *SMAX* be an implementation-defined <signed numeric literal> of declared type *DT*.

7) Case:

   a) If <sequence generator minvalue option> is specified, then

      Case:

      i) If NO MINVALUE is specified, then let *SMIN* be an implementation-defined <signed numeric literal> of declared type *DT*.

      ii) Otherwise, let *SMIN* be <sequence generator min value>.

   b) Otherwise, let *SMIN* be an implementation-defined <signed numeric literal> of declared type *DT*.

8) Case:

   a) If <sequence generator start with option> is specified, then let *START* be <sequence generator start value>.

   b) Otherwise,

Case:

i) If *SEQ* is an ascending sequence generator, then let *START* be *SMIN*.

ii) Otherwise, let *START* be *SMAX*.

9) The values of *INC*, *START*, *SMAX*, and *SMIN* shall all be exactly representable with the precision and scale of *DT*.

10) The value of *SMAX* shall be greater than the value of *SMIN*.

11) The value of *START* shall be greater than or equal to the value of *SMIN* and lesser than or equal to the value of *SMAX*.

12) If <sequence generator cycle option> is specified, then let *CYC* be <sequence generator cycle option>; otherwise, let *CYC* be NO CYCLE.

Access Rules

None.

General Rules

1) A sequence generator descriptor *SEQDS* that describes *SEQ* is created. *SEQDS* includes:

a) The sequence generator name that is a zero-length character string.

NOTE 235 — The name of an external sequence generator is later set by GR 1) of Subclause 11.62, "<sequence generator definition>"; however, internal sequence generators are anonymous.

b) The data type descriptor of *DT*.

c) The increment specified by *INC*.

d) The maximum value specified by *SMAX*.

e) The minimum value specified by *SMIN*.

f) The cycle option specified by *CYC*.

g) The current base value, set to *START*.

Conformance Rules

None.

9.25 Altering a sequence generator

Function

Complete the alteration of an internal or external sequence generator.

Syntax Rules

1) Let *OPT* and *SEQ* be the *OPTIONS* and *SEQUENCE* specified in an application of this Subclause. *OPT* shall conform to the Format of <alter sequence generator options>. The BNF nonterminal symbols used in the remainder of this Subclause refer to the contents of *OPT*.

2) Let *DT* be the data type descriptor included in *SEQ*.

3) Each of <alter sequence generator restart option>, <sequence generator increment by option>, <sequence generator maxvalue option>, <sequence generator minvalue option>, and <sequence generator cycle option> shall be specified at most once.

4) Case:

   a) If <sequence generator increment> is specified, then:

      i) Let *NEWIV* be <sequence generator increment>.

      ii) The value of *NEWIV* shall not be 0 (zero).

   b) Otherwise, let *NEWIV* be the increment of *SEQ*.

5) Case:

   a) If <sequence generator maxvalue option> is specified, then

      Case:

      i) If NO MAXVALUE is specified, then let *NEWMAX* be an implementation-defined <signed numeric literal> of declared type *DT*.

      ii) Otherwise, let *NEWMAX* be <sequence generator max value>.

   b) Otherwise let *NEWMAX* be the maximum value of *SEQ*.

6) Case:

   a) If <sequence generator minvalue option> is specified, then

      Case:

      i) If NO MINVALUE is specified, then let *NEWMIN* be an implementation-defined <signed numeric literal> of declared type *DT*.

      ii) Otherwise, let *NEWMIN* be <sequence generator min value>.

   b) Otherwise let *NEWMIN* be the minimum value of *SEQ*.

7) If <sequence generator cycle option> is specified, then let *NEWCYCLE* be <sequence generator cycle option>; otherwise, let *NEWCYCLE* be the cycle option of *SEQ*.

8) If <alter sequence generator restart option> is specified, then let *NEWVAL* be <sequence generator restart value>; otherwise, let *NEWVAL* be the current base value of *SEQ*.

9) The values of *NEWIV*, *NEWMAX*, *NEWMIN*, and *NEWVAL* shall all be exactly representable with the precision and scale of *DT*.

10) The value of *NEWMIN* shall be less than the value of *NEWMAX*.

11) The value of *NEWVAL* shall be greater than or equal to the value of *NEWMIN* and lesser than or equal to the value of *NEWMAX*.

Access Rules

None.

General Rules

1) *SEQ* is modified as follows:

   a) The increment is set to *NEWIV*.

   b) The maximum value is set to *NEWMAX*.

   c) The minimum value is set to *NEWMIN*.

   d) The cycle option is set to *NEWCYCLE*.

   e) The current base value is set to *NEWVAL*.

Conformance Rules

None.

(Blank page)

10 Additional common elements

This Clause is modified by Clause 8, "Additional common elements", in ISO/IEC 9075-4.
This Clause is modified by Clause 10, "Additional common elements", in ISO/IEC 9075-9.
This Clause is modified by Clause 7, "Additional common elements", in ISO/IEC 9075-10.
This Clause is modified by Clause 8, "Additional common elements", in ISO/IEC 9075-13.
This Clause is modified by Clause 11, "Additional common elements", in ISO/IEC 9075-14.

10.1 <interval qualifier>

Function

Specify the precision of an interval data type.

Format

```
<interval qualifier> ::=
    <start field> TO <end field>
  | <single datetime field>

<start field> ::=
  <non-second primary datetime field>
      [ <left paren> <interval leading field precision> <right paren> ]

<end field> ::=
    <non-second primary datetime field>
  | SECOND [ <left paren> <interval fractional seconds precision> <right paren> ]

<single datetime field> ::=
    <non-second primary datetime field>
        [ <left paren> <interval leading field precision> <right paren> ]
  | SECOND [ <left paren> <interval leading field precision>
      [ <comma> <interval fractional seconds precision> ] <right paren> ]

<primary datetime field> ::=
    <non-second primary datetime field>
  | SECOND

<non-second primary datetime field> ::=
    YEAR
  | MONTH
  | DAY
  | HOUR
  | MINUTE

<interval fractional seconds precision> ::=
  <unsigned integer>
```

```
<interval leading field precision> ::=
  <unsigned integer>
```

Syntax Rules

1) There is an ordering of significance of <primary datetime field>s. In order from most significant to least significant, the ordering is: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. A <start field> or <single datetime field> with an <interval leading field precision> *i* is more significant than a <start field> or <single datetime field> with an <interval leading field precision> *j* if *i*>*j*. An <end field> or <single datetime field> with an <interval fractional seconds precision> *i* is less significant than an <end field> or <single datetime field> with an <interval fractional seconds precision> *j* if *i*>*j*.

2) If TO is specified, then <start field> shall be more significant than <end field> and <start field> shall not specify MONTH. If <start field> specifies YEAR, then <end field> shall specify MONTH.

3) The maximum value of <interval leading field precision> is implementation-defined, but shall not be less than 2.

4) The maximum value of <interval fractional seconds precision> is implementation-defined, but shall not be less than 6.

5) An <interval leading field precision>, if specified, shall be greater than 0 (zero) and shall not be greater than the implementation-defined maximum. If <interval leading field precision> is not specified, then an <interval leading field precision> of 2 is implicit.

6) An <interval fractional seconds precision>, if specified, shall be greater than or equal to 0 (zero) and shall not be greater than the implementation-defined maximum. If SECOND is specified and <interval fractional seconds precision> is not specified, then an <interval fractional seconds precision> of 6 is implicit.

7) The precision of a field other than the <start field> or <single datetime field> is

   Case:

   a) If the field is not SECOND, then 2.

   b) Otherwise, 2 digits before the decimal point and the explicit or implicit <interval fractional seconds precision> after the decimal point.

Access Rules

None.

General Rules

1) An item qualified by an <interval qualifier> contains the datetime fields identified by the <interval qualifier>.

   Case:

   a) If the <interval qualifier> specifies a <single datetime field>, then the <interval qualifier> identifies a single <primary datetime field>. Any reference to the *most significant* or *least significant* <primary datetime field> of the item refers to that <primary datetime field>.

b) Otherwise, the <interval qualifier> identifies those datetime fields from <start field> to <end field>, inclusive.

2) An <interval leading field precision> specifies

Case:

a) If the <primary datetime field> is SECOND, then the number of decimal digits of precision before the specified or implied decimal point of the seconds <primary datetime field>.

b) Otherwise, the number of decimal digits of precision of the first <primary datetime field>.

3) An <interval fractional seconds precision> specifies the number of decimal digits of precision following the specified or implied decimal point in the <primary datetime field> SECOND.

4) The length in positions of an item of type interval is computed as follows.

Case:

a) If the item is a year-month interval, then

Case:

i) If the <interval qualifier> is a <single datetime field>, then the length in positions of the item is the implicit or explicit <interval leading field precision> of the <single datetime field>.

ii) Otherwise, the length in positions of the item is the implicit or explicit <interval leading field precision> of the <start field> plus 2 (the length of the <non-second primary datetime field> that is the <end field>) plus 1 (one) (the length of the <minus sign> between the <years value> and the <months value> in a <year-month literal>).

b) Otherwise,

Case:

i) If the <interval qualifier> is a <single datetime field> that does not specify SECOND, then the length in positions of the item is the implicit or explicit <interval leading field precision> of the <single datetime field>.

ii) If the <interval qualifier> is a <single datetime field> that specifies SECOND, then the length in positions of the item is the implicit or explicit <interval leading field precision> of the <single datetime field> plus the implicit or explicit <interval fractional seconds precision>. If <interval fractional seconds precision> is greater than zero, then the length in positions of the item is increased by 1 (one) (the length in positions of the <period> between the <seconds integer value> and the <seconds fraction>).

iii) Otherwise, let *participating datetime fields* mean the datetime fields that are less significant than the <start field> and more significant than the <end field> of the <interval qualifier>. The length in positions of each participating datetime field is 2.

Case:

1) If <end field> is SECOND, then the length in positions of the item is the implicit or explicit <interval leading field precision>, plus 3 times the number of participating datetime fields (each participating datetime field has length 2 positions, plus the <minus sign>s or <colon>s that precede them have length 1 (one) position), plus the implicit or explicit <interval fractional seconds precision>, plus 3 (the length in positions of the <end field> other than any

<interval fractional seconds precision> plus the length in positions of its preceding <colon>). If <interval fractional seconds precision> is greater than zero, then the length in positions of the item is increased by 1 (one) (the length in positions of the <period> within the field identified by the <end field>).

2) Otherwise, the length in positions of the item is the implicit or explicit <interval leading field precision>, plus 3 times the number of participating datetime fields (each participating datetime field has length 2 positions, plus the <minus sign>s or <colon>s that precede them have length 1 (one) position), plus 2 (the length in positions of the <end field>), plus 1 (one) (the length in positions of the <colon> preceding the <end field>).

Conformance Rules

1) Without Feature F052, “Intervals and datetime arithmetic”, conforming SQL language shall not contain an <interval qualifier>.

10.2 <language clause>

This Subclause is modified by Subclause 8.4, "<language clause>", in ISO/IEC 9075-13.

Function

Specify a standard programming language.

Format

```
<language clause> ::=
  LANGUAGE <language name>

<language name> ::=
    ADA
  | C
  | COBOL
  | FORTRAN
  | M | MUMPS
  | PASCAL
  | PLI
  | SQL
```

Syntax Rules

1) If MUMPS is specified, then M is implicit.

Access Rules

None.

General Rules

1) The standard programming language specified by the <language clause> is defined in the International Standard identified by the <language name> keyword. Table 15, "Standard programming languages", specifies the relationship.

Table 15 — Standard programming languages

Language keyword	Relevant standard
ADA	ISO/IEC 8652
C	ISO/IEC 9899
COBOL	ISO 1989

Language keyword	Relevant standard
FORTRAN	ISO 1539
M	ISO/IEC 11756
PASCAL	ISO/IEC 7185 and ISO/IEC 10206
PLI	ISO 6160
SQL	ISO/IEC 9075

Conformance Rules

None.

10.3 <path specification>

Function

Specify an order for searching for an SQL-invoked routine.

Format

```
<path specification> ::=
  PATH <schema name list>

<schema name list> ::=
  <schema name> [ { <comma> <schema name> }... ]
```

Syntax Rules

1) No two <schema name>s contained in <schema name list> shall be equivalent.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <path specification>.

10.4 <routine invocation>

This Subclause is modified by Subclause 8.1, "<routine invocation>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 7.1, "<routine invocation>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 8.3, "<routine invocation>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 11.1, "<routine invocation>", in ISO/IEC 9075-14.

Function

Invoke an SQL-invoked routine.

Format

```
<routine invocation> ::=
  <routine name> <SQL argument list>

<routine name> ::=
  [ <schema name> <period> ] <qualified identifier>

<SQL argument list> ::=
  <left paren> [ <SQL argument> [ { <comma> <SQL argument> }... ] ] <right paren>

<SQL argument> ::=
    <value expression>
  | <generalized expression>
  | <target specification>

<generalized expression> ::=
  <value expression> AS <path-resolved user-defined type name>
```

Syntax Rules

1) Let *RI* be the *ROUTINE INVOCATION*, let *TP* be the *SQLPATH* (if any), and let *UDTSM* be the *UDT* (if any) specified in an application of this Subclause.

2) Let *RN* be the <routine name> immediately contained in *RI*.

3) If *RI* is immediately contained in a <call statement>, then the <SQL argument list> of *RI* shall not contain a <generalized expression> without an intervening <routine invocation>.

4) Case:

   a) If *RI* is immediately contained in a <call statement>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* (henceforth, simply "possibly candidate routine") if *R* is an SQL-invoked procedure and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

   b) If *RI* is immediately contained in a <method selection>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an instance SQL-invoked method and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

c) If *RI* is immediately contained in a <constructor method selection>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an SQL-invoked constructor method and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

d) If *RI* is immediately contained in a <static method selection>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is a static SQL-invoked method and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN* and the method specification descriptor for *R* is included in a user-defined type descriptor for *UDTSM* or for some supertype of *UDTSM*.

e) Otherwise, an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an SQL-invoked regular function and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

5) Case:

a) If *RI* is contained in an <SQL schema statement>, then an <SQL-invoked routine> *R* is an *executable routine* if and only if *R* is a possibly candidate routine and the applicable privileges for the <authorization identifier> that owns the containing schema include EXECUTE on *R*.

b) Otherwise, an <SQL-invoked routine> *R* is an *executable routine* if and only if *R* is a possibly candidate routine and the current privileges include EXECUTE on *R*.

6) Case:

a) If <SQL argument list> does not immediately contain at least one <SQL argument>, then an *invocable routine* is an executable routine that has no SQL parameters.

b) Otherwise:

i) Let *NA* be the number of <SQL argument>s in the <SQL argument list> *AL* of *RI*. Let A_i, 1 (one) $\leq i \leq NA$, be the *i*-th <SQL argument> in *AL*.

ii) Let the *static SQL argument list* of *RI* be *AL*.

iii) Let P_i be the *i*-th SQL parameter of an executable routine. An *invocable routine* is an SQL-invoked routine *SIR* that is an executable routine such that:

1) *SIR* has *NA* SQL parameters.

2) If *RI* is not immediately contained in a <call statement>, then for each A_i that is not a <dynamic parameter specification>,

Case:

A) If the declared type of P_i is a user-defined type, then:

I) Let ST_i be the set of subtypes of the declared type of A_i.

II) The type designator of the declared type of P_i shall be in the type precedence list of the data type of some type in ST_i.

NOTE 236 — "type precedence list" is defined in Subclause 9.5, "Type precedence list determination".

B) Otherwise, the type designator of the declared type of P_i shall be in the type precedence list of the declared type of A_i.

NOTE 237 — "type precedence list" is defined in Subclause 9.5, "Type precedence list determination".

7) If <SQL argument list> does not immediately contain at least one <SQL argument>, then:

a) Let *AL* be an empty list of SQL arguments.

b) The subject routine of *RI* is defined as follows:

i) If *RN* does not contain a <schema name>, then:

1) Case:

A) If *RI* is immediately contained in a <method selection>, <static method selection>, or a <constructor method selection>, then let *DP* be *TP*.

B) If the routine execution context of the current SQL-session indicates that an SQL-invoked routine is active, then let *DP* be the routine SQL-path of that routine execution context.

C) Otherwise,

Case:

I) If *RI* is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

II) If *RI* is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then let *DP* be the SQL-path of the current SQL-session.

III) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains *RI*.

2) The *subject routine* of *RI* is an SQL-invoked routine *SIRSR* such that:

A) *SIRSR* is an invocable routine.

B) The <schema name> of the schema of *SIRSR* is in *DP*.

C) Case:

I) If the routine descriptor of *SIRSR* does not include a STATIC indication, then there is no other invocable routine *R2* for which the the <schema name> of the schema that includes *R2* precedes in *DP* the <schema name> of the schema that includes *SIRSR*.

II) If the routine descriptor of *SIRSR* includes a STATIC indication, then there is no other invocable routine *R2* for which the user-defined type described by the descriptor that includes the routine descriptor of *R2* is a subtype of the user-defined type described by the user-defined type descriptor that includes the routine descriptor of *SIRSR*.

ii) If *RN* contains a <schema name> *SN*, then *SN* shall be the <schema name> of a schema *S*. The *subject routine* of *RI* is the invocable routine (if any) contained in *S*.

c) There shall be exactly one subject routine of *RI*.

d) If *RI* is not immediately contained in a <call statement>, then the *effective returns data type* of *RI* is the result data type of the subject routine of *RI*.

e) Let the *static SQL argument list* of *RI* be an empty list of SQL arguments.

8) If <SQL argument list> immediately contains at least one <SQL argument>, then:

a) The <data type> of each <value expression> immediately contained in a <generalized expression> shall be a subtype of the structured type identified by the <user-defined type name> simply contained in the <path-resolved user-defined type name> that is immediately contained in <generalized expression>.

b) The set of *candidate routines* of *RI* is defined as follows.

Case:

i) If *RN* does not contain a <schema name>, then:

1) Case:

A) If *RI* is immediately contained in a <method selection>, a <static method selection>, or a <constructor method selection>, then let *DP* be *TP*.

B) If the routine execution context of the current SQL-session indicates that an SQL-invoked routine is active, then let *DP* be the routine SQL-path of that routine execution context.

C) Otherwise,

Case:

I) If *RI* is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

II) If *RI* is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then let *DP* be the SQL-path of the current SQL-session.

III) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains *RI*.

2) The candidate routines of *RI* are the set union of invocable routines of all schemas whose <schema name> is in *DP*.

ii) If *RN* contains a <schema name> *SN*, then *SN* shall be the <schema name> of a schema *S*. The candidate routines of *RI* are the invocable routines (if any) contained in *S*.

c) Case:

i) If *RI* is immediately contained in a <call statement>, then:

1) Let *XAL* be *AL*.

2) The subject routine *SR* of *XAL* is the SQL-invoked routine *SIRCR1* that is a candidate routine of *RI* such that there is no other candidate routine *R2* for which the <schema name> of the schema that includes *R2* precedes in *DP* the <schema name> of the schema that includes *SIRCR1*.

3) Let *PL* be the list of SQL parameters P_i of *SR*.

4) For each P_i that is an output SQL parameter or both an input SQL parameter and an output SQL parameter, A_i shall be a <target specification>.

   A) If *RI* is contained in a <triggered SQL statement> of an AFTER trigger, then A_i shall not be a <column reference>.

   B) If A_i is an <embedded variable specification> or a <host parameter specification>, then P_i shall be assignable to A_i, according to the Syntax Rules of Subclause 9.1, "Retrieval assignment", with A_i and P_i as *TARGET* and *VALUE*, respectively.

   C) If A_i is an <SQL parameter reference>, a <column reference>, or a <target array element specification>, then P_i shall be assignable to A_i, according to the Syntax Rules of Subclause 9.2, "Store assignment", with A_i and P_i as *TARGET* and *VALUE*, respectively.

   NOTE 238 — The <column reference> can only be a new transition variable column reference.

5) For each P_i that is an input SQL parameter but not an output SQL parameter, A_i shall be a <value expression>.

6) For each P_i that is an input SQL parameter or both an input SQL parameter and an output SQL parameter, A_i shall be assignable to P_i, according to the Syntax Rules of Subclause 9.2, "Store assignment", with P_i and A_i as *TARGET* and *VALUE*, respectively.

ii) Otherwise:

1) A_i shall be a <value expression> or <generalized expression>.

2) Case:

   A) If A_i is a <generalized expression>, then let TS_i be the data type identified by the <user-defined type name> simply contained in the <path-resolved user-defined type name> that is immediately contained in the <generalized expression>.

   B) Otherwise, let TS_i be the data type whose data type name is included in the data type descriptor of the data type of A_i.

3) The *subject routine* is defined as follows:

   A) For each A_i,

      Case:

      I) If A_i is a <dynamic parameter specification>, then let V_i be A_i.

      II) Otherwise, let V_i be a value arbitrarily chosen whose declared type is TS_i.

   B) Let *XAL* be an <SQL argument list> with *N* <SQL argument>s derived from the V_is ordered according to their ordinal position *i* in *XAL*. The Syntax Rules of Subclause 9.4, "Subject routine determination", are applied to the candidate routines of *RI* and *XAL*, yielding a set of candidate subject routines *CSR*.

   C) Case:

I) If *RN* contains a <schema name>, then there shall be exactly one candidate subject routine in *CSR*. The subject routine *SR* is the candidate subject routine in *CSR*.

II) Otherwise:

1) There shall be at least one candidate subject routine in *CSR*.

2) Case:

a) If there is exactly one candidate subject routine in *CSR*, then the subject routine *SR* is the candidate subject routine in *CSR*.

b) If there is more than one candidate subject routine in *CSR*, then

Case:

i) If *RI* is not immediately contained in a <static method selection>, then there shall be an SQL-invoked routine *SIRCR2* in *CSR* such that there is no other candidate subject routine *R2* in *CSR* for which any of the following is true:

1) The <schema name> of the schema that includes *R2* precedes in *DP* the <schema name> of the schema that includes *SIRCR2*.

2) The <schema name> of the schema that includes *R2* is equivalent to the <schema name> of the schema that includes *SIRCR2*.

The subject routine *SR* is *SIRCR2*.

ii) Otherwise, there shall be an SQL-invoked routine *SIRCR3* in *CSR* such that there is no other candidate subject routine *R2* in *CSR* for which the user-defined type described by the user-defined type descriptor that includes the routine descriptor of *R2* is a subtype of the user-defined type described by the user-defined type descriptor that includes the routine descriptor of *SIRCR3*. The subject routine *SR* is *SIRCR3*.

4) The subject routine of *RI* is the subject routine *SR*.

5) Let *PL* be the list of SQL parameters P_i of *SR*.

6) For each P_i, A_i shall be assignable to P_i according to the Syntax Rules of Subclause 9.2, "Store assignment", with P_i as *TARGET* and A_i as *VALUE*.

7) The *effective returns data type* of *RI* is defined as follows:

A) Case:

I) If *SR* is a type-preserving function, then let P_i be the result SQL parameter of *SR*. If A_i contains a <generalized expression>, then let *RT* be the declared type of the <value expression> contained in the <generalized expression> of A_i; otherwise, let *RT* be the declared type of A_i.

II) Otherwise, let *RT* be the result data type of *SR*.

B) The *effective returns data type* of *RI* is *RT*.

9) If *SR* is a constructor function, then *RI* shall be simply contained in a <new invocation>.

Access Rules

None.

General Rules

1) Let *SAL* be the *STATIC SQL ARG LIST* and let *SR* be the *SUBJECT ROUTINE* specified in an application of this Subclause.

NOTE 239 — "static SQL argument list" and "subject routine" are defined by the Syntax Rules of this Subclause.

2) Case:

a) If *SAL* is empty, then let the *dynamic SQL argument list DAL* be *SAL*.

b) Otherwise:

i) Each SQL argument A_i in *SAL* is evaluated, in an implementation-dependent order, to obtain a value V_i.

ii) Let the *dynamic SQL argument list DAL* be the list of values V_i in order.

iii) If *SR* is type preserving and the null value is substituted for the result parameter, then

Case:

1) If *SR* is a mutator function, then an exception condition is raised: *data exception — null value substituted for mutator subject parameter*.

2) Otherwise, the value of *RI* is the null value and the remaining General Rules of this Subclause are not applied.

iv) Case:

1) If *SR* is an instance SQL-invoked method, then:

A) If V_1 is the null value, then the value of *RI* is the null value and the remaining General Rules of this Subclause are not applied.

B) Let *SM* be the set of SQL-invoked methods *M* that satisfy the following conditions:

I) The <routine name> of *SR* and the <routine name> of *M* have equivalent <qualified identifier>s.

II) *SR* and *M* have the name number *N* of SQL parameters. Let PSR_i, 1 (one) $\le i \le N$, be the *i*-th SQL parameter of *SR* and PM_i, 1 (one) $\le i \le N$, be the *i*-th SQL parameter of *M*.

III) The declared type of the subject parameter of *M* is a subtype of the declared type of the subject parameter of *SR*.

IV) For j varying from 2 to N, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with the declared type of PM_j and the declared type of PSR_j.

NOTE 240 — *SR* is an element of the set *SM*.

C) *SM* is the *set of overriding methods* of *SR* and every SQL-invoked method *M* in *SM* is an *overriding method* of *SR*.

D) Case:

I) If the first SQL argument *A1* in *SAL* contains a <generalized expression>, then let *DT1* be the data type identified by the <user-defined type name> contained in the <generalized expression> of *A1*.

II) Otherwise, let *DT1* be the most specific type of V_1.

E) Let *R* be the SQL-invoked method in *SM* such that there is no other SQL-invoked method *M1* in *SM* for which the type designator of the declared type of the subject parameter of *M1* precedes that of the declared type of the subject parameter of *R* in the type precedence list of *DT1*.

2) Otherwise, let *R* be *SR*.

3) Let *N* and *PN* be the number of values V_i in *DAL*. Let T_i be the declared type of the *i*-th SQL parameter P_i of *R*. For *i* ranging from 1 (one) to *PN*,

Case:

a) If P_i is an input SQL parameter or both an input SQL parameter and an output SQL parameter, then let CPV_i be the result an application of the General Rules of Subclause 9.2, "Store assignment", with V_i as *VALUE* and T_i as *TARGET*.

b) Otherwise,

Case:

i) If *R* is an SQL routine, then let CPV_i be the null value.

ii) Otherwise, let CPV_i be an implementation-defined value of most specific type T_i.

4) If *R* is an external routine, then:

a) Let *P* be the program identified by the external name of *R*.

b) For *i* ranging from 1 (one) to *N*, let P_i be the *i*-th SQL parameter of *R* and let T_i be the declared type of P_i.

Case:

i) If P_i is an input SQL parameter or both an input SQL parameter and an output SQL parameter, then

Case:

1) If P_i is a locator parameter, then CPV_i is replaced by the locator value that uniquely identifies the value of CPV_i.

2) If T_i is a user-defined type, and P_i is not a locator parameter, then:

A) Let FSF_i be the SQL-invoked routine identified by the specific name of the from-sql function associated with P_i in the routine descriptor of *R*. Let RT_i be the result data type of FSF_i.

B) The General Rules of this Subclause are applied with a static SQL argument list that has a single argument that is CPV_i and subject routine FSF_i.

C) Let RV_i be the result of the invocation of FSF_i. CPV_i is replaced by RV_i.

ii) Otherwise,

Case:

1) If P_i is a locator parameter, then CPV_i is replaced with an implementation-dependent value of type INTEGER.

2) If T_i is a user-defined type and P_i is not a locator parameter, then:

A) Let FSF_i be the SQL-invoked routine identified by the specific name of the from-sql function associated with P_i in the routine descriptor of *R*. Let RT_i be the result data type of FSF_i.

B) CPV_i is replaced by an implementation-defined value of type RT_i.

5) Preserve the current SQL-session context *CSC* and create a new SQL-session context *RSC* derived from *CSC* as follows:

a) Set the current default catalog name, the current default unqualified schema name, the current default character set name, the SQL-path of the current SQL-session, the current default time zone displacement of the current SQL-session, and the contents of all SQL dynamic descriptor areas to implementation-defined values.

b) Set the values of the current SQL-session identifier, the identities of all instances of global temporary tables, the current constraint mode for each integrity constraint, the current access mode, the current isolation level, and the current condition area limit to their values in *CSC*.

c) The diagnostics area stack in *CSC* is copied to *RSC* and the General Rules of Subclause 23.2, "Pushing and popping the diagnostics area stack", are applied with "PUSH" as *OPERATION* and the diagnostics area stack in *RSC* as *STACK*.

d) Case:

i) If *R* is an SQL routine, then remove from *RSC* the identities of all instances of created local temporary tables, declared local temporary tables that are defined by <temporary table declaration>s that are contained in <SQL-client module definition>s, and the cursor instance descriptors of all open cursors that are not global extended dynamic cursors.

ii) Otherwise:

1) Remove from *RSC* the identities of all instances of created local temporary tables that are referenced in <SQL-client module definition>s that are not the <SQL-client module definition> of *P*, declared local temporary tables that are defined by <temporary table declaration>s that are contained in <SQL-client module definition>s that are not the <SQL-client module

definition> of *P*, and the cursor instance descriptors of all open cursors that are not global extended dynamic cursors and whose SQL-client module is not the SQL-client module of *P*.

2) It is implementation-defined whether the identities of all instances of created local temporary tables that are referenced in the <SQL-client module definition> of *P*, declared local temporary tables that are defined by <temporary table declaration>s that are contained in the <SQL-client module definition> of *P*, and the cursor instance descriptors of all open cursors that are not global extended dynamic cursors and SQL-client module is the SQL-client module of *P* are removed from *RSC*.

e) Indicate in the routine execution context of *RSC* that the SQL-invoked routine *R* is active.

f) Case:

i) If the SQL-data access indication of *CSC* specifies possibly contains SQL and *R* possibly reads SQL-data or *R* possibly modifies SQL-data, then:

1) If *R* is an external routine, then an exception condition is raised: *external routine exception — reading SQL-data not permitted*.

2) Otherwise, an exception condition is raised: *SQL routine exception — reading SQL-data not permitted*.

ii) If the SQL-data access indication of *CSC* specifies possibly reads SQL and *R* possibly modifies SQL-data, then:

1) If *R* is an external routine, then an exception condition is raised: *external routine exception — modifying SQL-data not permitted*.

2) Otherwise, an exception condition is raised: *SQL routine exception — modifying SQL-data not permitted*.

g) Case:

i) If *R* does not possibly contain SQL, then set the SQL-data access indication in the routine execution context of *RSC* to *does not possibly contain SQL*.

ii) If *R* possibly contains SQL, then set the SQL-data access indication in the routine execution context of *RSC* to *possibly contains SQL*.

iii) If *R* possibly reads SQL-data, then set the SQL-data access indication in the routine execution context of *RSC* to *possibly reads SQL-data*.

iv) If *R* possibly modifies SQL-data, then set the SQL-data access indication in the routine execution context of *RSC* to *possibly modifies SQL-data*.

h) The authorization stack of *RSC* is set to a copy of the authorization stack of *CSC*.

i) A copy of the top cell is pushed onto the authorization stack of *RSC*.

j) Case:

i) If *R* is an external routine, then:

1) Case:

A) If the external security characteristic of *R* is IMPLEMENTATION DEFINED, then the current user identifier and the current role name of *RSC* are implementation-defined.

B) If the external security characteristic of *R* is DEFINER, then the top cell of the authorization stack of *RSC* is set to contain only the routine authorization identifier of *R*.

2) The routine SQL-path of *RSC* is set to the external routine SQL-path of *R*.

ii) Otherwise:

1) If the SQL security characteristic of *R* is DEFINER, then the top cell of the authorization stack of *RSC* is set to contain only the routine authorization identifier of *R*.

2) The routine SQL-path of *RSC* is set to the routine SQL-path of *R*.

k) If the subject routine is an SQL-invoked procedure *SIP*, then:

i) For every received cursor *RC* whose origin is the <specific routine designator> of *SIP*, the cursor declaration descriptor and the cursor instance descriptor of *RC* are destroyed.

ii) Every result set sequence associated with *SIP* is destroyed.

iii) Let *INV* be the invoker of *SIP*. An empty result set sequence *RSS*, for SQL-invoked procedure *SIP* and invoker *INV*, is added to *RSC*.

NOTE 241 — The invoker of SIP is defined in Subclause 4.27.5, "Result sets returned by SQL-invoked procedures".

l) *RSC* becomes the current SQL-session context.

6) If the descriptor of *R* includes an indication that a new savepoint level is to be established when *R* is invoked, then a new savepoint level is established.

7) If *R* is an SQL routine, then

Case:

a) If *R* is a null-call function and if any of CPV_i is the null value, then let *RV* be the null value.

b) Otherwise:

i) For *i* ranging from 1 (one) to *PN*, set the value of P_i to CPV_i.

ii) The General Rules of Subclause 13.5, "<SQL procedure statement>", are evaluated with the SQL routine body of *R* as the *executing statement*.

iii) If, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-connection statement, then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted*.

iv) Case:

1) If the SQL-implementation does not support Feature T272, "Enhanced savepoint management", and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-transaction statement, then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted*.

2) If, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-transaction statement that is not a <savepoint statement> or a <release savepoint statement>, or is a <rollback statement> that does not specify a <savepoint clause>,

then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted*.

v) If the SQL implementation does not support Feature T651, "SQL-schema statements in SQL routines", and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-schema statement, an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted*.

vi) If the SQL implementation does not support Feature T652, "SQL-dynamic statements in SQL routines", and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-dynamic statement, an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted*.

vii) If the SQL-data access indication of *RSC* specifies *possibly contains SQL* and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-statement that possibly reads SQL-data, or an attempt is made to execute an SQL-statement that possibly modifies SQL-data, then an exception condition is raised: *SQL routine exception — reading SQL-data not permitted*.

viii) If the SQL-data access indication of *RSC* specifies *possibly reads SQL-data* and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-statement that possibly modifies SQL-data then an exception condition is raised: *SQL routine exception — modifying SQL-data not permitted*.

ix) If *R* is an SQL-invoked function, then

Case:

1) If no <return statement> is executed before completion of the execution of the SQL routine body of *R*, then an exception condition is raised: *SQL routine exception — function executed no return statement*.

2) Otherwise, let *RV* be the returned value of the execution of the SQL routine body of *R*.

NOTE 242 — "Returned value" is defined in Subclause 16.2, "<return statement>".

x) If *R* is an SQL-invoked procedure, then for each SQL parameter of *R* that is an output SQL parameter or both an input SQL parameter and an output SQL parameter, set the value of CPV_i to the value of P_i.

8) If *R* is an external routine, then:

a) The method and time of binding of *P* to the schema or SQL-server module that includes *R* is implementation-defined.

b) If *R* specifies PARAMETER STYLE SQL, then

i) Case:

1) If *R* is an SQL-invoked function, then the effective SQL parameter list *ESPL* of *R* is set as follows:

A) If *R* is an array-returning external function or a multiset-returning external function with the element type being a row type, then let *FRN* be the degree of the element type; otherwise, let *FRN* be 1 (one).

B) For *i* ranging from 1 (one) to *PN*, the *i*-th entry in *ESPL* is set to CPV_i.

C) For *i* ranging from *PN*+1 to *PN*+*FRN*, the *i*-th entries in *ESPL* are the *result data items*.

D) For *i* ranging from (*PN*+*FRN*)+1 to (*PN*+*FRN*)+*N*, the *i*-th entry in *ESPL* is the *SQL indicator argument* corresponding to $CPV_{i-(PN+FRN)}$.

E) For *i* ranging from (*PN*+*FRN*)+*N*+1 to (*PN*+*FRN*)+*N*+*FRN*, the *i*-th entries in *ESPL* are the SQL indicator arguments corresponding to the result data items.

F) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+1, the *i*-th entry in *ESPL* is the *exception data item*.

G) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+2, the *i*-th entry in *ESPL* is the *routine name text item*.

H) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+3, the *i*-th entry in *ESPL* is the *specific name text item*.

I) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+4, the *i*-th entry in *ESPL* is the *message text item*.

J) If *R* is an array-returning external function or a multiset-returning external function, then for *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+5, the *i*-th entry in *ESPL* is the *save area data item* and for *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+6, the *i*-th entry in *ESPL* is the *call type data item*.

K) Set the values of the SQL indicator arguments corresponding to the result data items (that is, SQL argument value list entries from (*PN*+*FRN*)+*N*+1 through (*PN*+*FRN*)+*N*+*FRN*, inclusive, to 0 (zero).

L) For *i* ranging from 1 (one) to *PN*, if CPV_i is the null value, then set entry (*PN*+*FRN*)+*i* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to –1; otherwise, set entry (*PN*+*FRN*)+*i* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to 0 (zero).

M) If *R* is an array-returning external function or a multiset-returning external function, then set the value of the save area data item (that is, SQL argument value list entry (*PN*+*FRN*)+(*N*+*FRN*)+5) to 0 (zero) and set the value of the call type data item (that is, SQL argument value list entry (*PN*+*FRN*)+(*N*+*FRN*)+6) to –1.

2) Otherwise, the effective SQL parameter list *ESPL* of *R* is set as follows:

A) For *i* ranging from 1 (one) to *PN*, the *i*-th entry in *ESPL* is CPV_i.

B) For *i* ranging from *PN*+1 to *PN*+*N*, the *i*-th entry in *ESPL* is the *SQL indicator argument* corresponding to CPV_{i-PN}.

C) For *i* equal to (*PN*+*N*)+1, the *i*-th entry in *ESPL* is the *exception data item*.

D) For *i* equal to (*PN*+*N*)+2, the *i*-th entry in *ESPL* is the *routine name text item*.

E) For *i* equal to (*PN*+*N*)+3, the *i*-th entry in *ESPL* is the *specific name text item*.

F) For *i* equal to (*PN*+*N*)+4, the *i*-th entry in *ESPL* is the *message text item*.

G) For *i* ranging from 1 (one) to *PN*, if CPV_i is the null value, then set entry *PN*+*i* in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to –1; otherwise, set entry *PN*+*i* in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to 0 (zero).

ii) The exception data item is set to '00000'.

iii) The routine name text item is set to the <schema qualified name> of the routine name of *R*.

iv) The specific name text item is set to the <qualified identifier> of the specific name of *R*.

v) The message text item is set to a zero-length string.

c) If *R* specifies PARAMETER STYLE GENERAL, then the effective SQL parameter list *ESPL* of *R* is set as follows:

i) If *R* is not a null-call function and, for *i* ranging from 1 (one) to *PN*, CPV_i is the null value, then an exception condition is raised: *external routine invocation exception — null value not allowed*.

ii) For *i* ranging from 1 (one) to *PN*, if no CPV_i is the null value, then the for *j* ranging from 1 (one) to *PN*, if the *j*-th entry in *ESPL* is set to CPV_j.

d) If *R* specifies DETERMINISTIC and if different executions of *P* with identical SQL argument value lists do not produce identical results, then the results are implementation-dependent.

e) Let *EN* be the number of entries in *ESPL*. Let ESP_i be the *i*-th effective SQL parameter in *ESPL*.

f) Case:

i) If *R* is a null-call function and if any of CPV_i is the null value, then *P* is assumed to have been executed.

ii) Otherwise:

1) If *R* is not an array-returning external function or a multiset-returning external function, then *P* is executed with a list of *EN* parameters PD_i whose parameter names are PN_i and whose values are set as follows:

A) Depending on whether the language of *R* specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the *operative data type correspondences table* be Table 16, “Data type correspondences for Ada”, Table 17, “Data type correspondences for C”, Table 18, “Data type correspondences for COBOL”, Table 19, “Data type correspondences for Fortran”, Table 20, “Data type correspondences for M”, Table 21, “Data type correspondences for Pascal”, or Table 22, “Data type correspondences for PL/I”, respectively. Refer to the two columns of the operative data type correspondences table as the “SQL data type” column and the “host data type” column.

B) For *i* varying from 1 (one) to *EN*, the data type DT_i of PD_i is the data type listed in the host data type column of the row in the data type correspondences table whose value in the SQL data type column corresponds to the data type of ESP_i.

C) The value of PD_i is set to the value of ESP_i.

2) If *R* is an array-returning external function, then:

A) Let *AR* be an array whose declared type is the result data type of *R*.

B) The General Rules of Subclause 9.14, “Execution of array-returning functions”, are applied with *AR*, *ESPL*, and *P* as *ARRAY*, *EFFECTIVE SQL PARAMETER LIST*, and *PROGRAM*, respectively.

3) If *R* is a multiset-returning external function, then:

   A) Let *MU* be a multiset whose declared type is the result data type of *R*.

   B) The General Rules of Subclause 9.15, "Execution of multiset-returning functions", are applied with *MU*, *ESPL*, and *P* as *MULTISET*, *EFFECTIVE SQL PARAMETER LIST*, and *PROGRAM*, respectively.

4) If the SQL-data access indication of *RSC* specifies *does not possibly contain SQL* and, before the completion of any execution of *P*, an attempt is made to execute an SQL-statement, then an exception condition is raised: *external routine exception — containing SQL not permitted*.

5) If, before the completion of any execution of *P*, an attempt is made to execute an SQL-connection statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted*.

6) Case:

   A) If the SQL-implementation does not support Feature T272, "Enhanced savepoint management", and, before the completion of the execution of *P*, an attempt is made to execute an SQL-transaction statement, then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted*.

   B) If, before the completion of the execution of *P*, an attempt is made to execute an SQL-transaction statement that is not <savepoint statement> or <release savepoint statement>, or is a <rollback statement> that does not specify a <savepoint clause>, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted*.

7) If the SQL implementation does not support Feature T653, "SQL-schema statements in external routines", and, before the completion of any execution of *P*, an attempt is made to execute an SQL-schema statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted*.

8) If the SQL implementation does not support Feature T654, "SQL-dynamic statements in external routines", and, before the completion of any execution of *P*, an attempt is made to execute an SQL-dynamic statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted*.

9) If the SQL-data access indication of *RSC* specifies *possibly contains SQL* and, before the completion of any execution of *P*, an attempt is made to execute an SQL-statement that possibly reads SQL-data, or an attempt is made to execute an SQL-statement that possibly modifies SQL-data, then an exception condition is raised: *external routine exception — reading SQL-data not permitted*.

10) If the SQL-data access indication of *RSC* specifies *possibly reads SQL* and, before the completion of any execution of *P*, an attempt is made to execute an SQL-statement that possibly modifies SQL-data, then an exception condition is raised: *external routine exception — modifying SQL-data not permitted*.

11) If the language specifies ADA (respectively C, COBOL, FORTRAN, M, PASCAL, PLI) and *P* is not a standard-conforming Ada program (respectively C, COBOL, Fortran, M, Pascal, PL/I program), then the results of any execution of *P* are implementation-dependent.

g) After the completion of any execution of *P*:

   i) It is implementation-defined whether:

1) Every instance of created local temporary tables and every instance of declared local temporary tables that is associated with *RSC* is destroyed.

2) For every prepared statement *PS* prepared by *P* in the current SQL-transaction that has not been deallocated by *P*:

   A) Let *SSN* be the <SQL statement name> that identifies *PS*.

   B) The following SQL-statement is effectively executed:

```
DEALLOCATE PREPARE SSN
```

ii) For *i* varying from 1 (one) to *EN*, the value of ESP_i is set to the value of PD_i. If *R* specifies PARAMETER STYLE SQL, then

Case:

1) If the exception data item has the value '00000', then the execution of *P* was successful.

2) If the first two characters of the exception data item are equal to the SQLSTATE condition code class value for *warning*, then a completion condition is raised: *warning*, using a subclass code equal to the final three characters of the value of the exception data item.

3) Otherwise, an exception condition is raised using a class code equal to the first two characters of the value of the exception data item and a subclass code equal to the final three characters of the value of the exception data item.

iii) If the exception data item is not '00000' and *R* specified PARAMETER STYLE SQL, then the message text item is stored in the first diagnostics area.

h) If *R* is an SQL-invoked function, then:

i) Case:

1) If *R* is an SQL-invoked method whose routine descriptor does not include a STATIC indication and if CPV_1 is the null value, then let *RDI* be the null value.

2) If *R* is a null-call function, *R* is not an array-returning external function or a multiset-returning external function, and if any of CPV_i is the null value, then let *RDI* be the null value.

3) If *R* is not a null-call function, *R* is not an array-returning function or a multiset-returning function, *R* specifies PARAMETER STYLE SQL, and entry (*PN*+1)+*N*+1 in *ESPL* (that is, SQL indicator argument *N*+1 corresponding to the result data item) is negative, then let *RDI* be the null value.

4) Otherwise,

   A) Case:

      I) If *R* is not an array-returning external function or a multiset-returning external function, *R* specifies PARAMETER STYLE SQL, and entry (*PN*+1)+*N*+1 in *ESPL* (that is, SQL indicator argument *N*+1 corresponding to the result data item) is not negative, then let *ERDI* be the value of the result data item.

      II) If *R* is an array-returning external function, and *R* specifies PARAMETER STYLE SQL, then let *ERDI* be *AR*.

III) If *R* is a multiset-returning function, and *R* specifies PARAMETER STYLE SQL, then let *ERDI* be *MU*.

IV) If *R* specifies PARAMETER STYLE GENERAL, then let *ERDI* be the value returned from *P*.

NOTE 243 — The value returned from *P* is passed to the SQL-implementation in an implementation-dependent manner. An argument value list entry is not used for this purpose.

B) Case:

I) If the routine descriptor of *R* indicates that the return value is a locator, then

Case:

1) If *RT* is a binary large object type, then let *RDI* be the binary large object value corresponding to *ERDI*.

2) If *RT* is a character large object type, then let *RDI* be the large object character string corresponding to *ERDI*.

3) If *RT* is an array type, then let *RDI* be the array value corresponding to *ERDI*.

4) If *RT* is a multiset type, then let *RDI* be the multiset value corresponding to *ERDI*.

5) If *RT* is a user-defined type, then let *RDI* be the user-defined type value corresponding to *ERDI*.

II) Otherwise, if *R* specifies <result cast>, then let *CRT* be the <data type> specified in <result cast>; otherwise, let *CRT* be the <returns data type> of *R*.

Case:

1) If *R* specifies <result cast> and the routine descriptor of *R* indicates that the <result cast> has a locator indication, then

Case:

a) If *CRT* is a binary large object type, then let *RDI* be the binary large object value corresponding to *ERDI*.

b) If *CRT* is a character large object type, then let *RDI* be the large object character string corresponding to *ERDI*.

c) If *CRT* is an array type, then let *RDI* be the array value corresponding to *ERDI*.

d) If *CRT* is a multiset type, then let *RDI* be the multiset value corresponding to *ERDI*.

e) If *CRT* is a user-defined type, then let *RDI* be the user-defined type value corresponding to *ERDI*.

2) Otherwise,

Case:

a) If *CRT* is a user-defined type, then:

i) Let *TSF* be the SQL-invoked routine identified by the specific name of the to-sql function associated with the result of *R*.

ii) Case:

1) If *TSF* is an SQL-invoked method, then:

A) If *R* is a type-preserving function, then let *MAT* be the most specific type of the value of the argument substituted for the result SQL parameter of *R*; otherwise, let *MAT* be *CRT*.

B) The General Rules of this Subclause are applied with a static SQL argument list whose first element is the value returned by the invocation of:

```
MAT()
```

and whose second element is *ERDI*, and the subject routine *TSF*.

2) Otherwise, the General Rules of this Subclause are applied with a static SQL argument list that has a single SQL argument that is *ERDI*, and the subject routine *TSF*.

iii) Let *RDI* be the result of invocation of *TSF*.

b) Otherwise, let *RDI* be *ERDI*.

ii) If *R* specified a <result cast>, then let *RT* be the <returns data type> of *R* and let *RV* be the result of:

```
CAST ( RDI AS RT )
```

Otherwise, let *RV* be *RDI*.

i) If *R* is an SQL-invoked procedure, then for each P_i, 1 (one) ≤ i ≤ *PN*, that is an output SQL parameter or both an input SQL parameter and an output SQL parameter,

Case:

i) If *R* specifies PARAMETER STYLE SQL and entry (*PN*+1)+*i* in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) is negative, then CPV_i is set to the null value.

ii) If *R* specifies PARAMETER STYLE SQL, and entry (*PN*+1)+*i* in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) is not negative, and a value was not assigned to the *i*-th entry in *ESPL*, then CPV_i is set to an implementation-defined value of type T_i.

iii) Otherwise:

NOTE 244 — In this case, either *R* specifies PARAMETER STYLE SQL and entry (*PN*+1)+*i* in *SQPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) is not negative and a value was assigned to the *i*-th entry in *ESPL*, or else *R* specifies PARAMETER STYLE GENERAL.

1) Let EV_i be the *i*-th entry in *ESPL*. Let T_i be the <data type> of P_i.

2) Case:

A) If P_i is a locator parameter, then

Case:

I) If T_i is a binary large object type, then CPV_i is set to the binary large object value corresponding to EV_i.

II) If T_i is a character large object type, then CPV_i is set to the large object character string corresponding to EV_i.

III) If T_i is an an array type, then CPV_i is set to the array value corresponding to EV_i.

IV) If T_i is an a multiset type, then CPV_i is set to the multiset value corresponding to EV_i.

V) If T_i is a user-defined type, then CPV_i is set to the user-defined type value corresponding to EV_i.

B) If T_i is a user-defined type, then:

I) Let TSF_i be the SQL-invoked function identified by the specific name of the to-sql function associated with P_i in the routine descriptor of R.

II) Case:

1) If *TSF* is an SQL-invoked method, then the General Rules of this Subclause are applied with a static SQL argument list whose first element is the value returned by the invocation of:

```
Ti()
```

and whose second element is EV_i, and the subject routine TSF_i.

2) Otherwise, the General Rules of this Subclause are applied with a static SQL argument list that has a single SQL argument that is EV_i, and the subject routine TSF_i.

III) CPV_i is set to the result of an invocation of TSF_i.

C) Otherwise, CPV_i is set to EV_i.

9) Case:

a) If *R* is an SQL-invoked function, then:

i) If *R* is a type-preserving function, then:

1) Let *MAT* be the most specific type of the value of the argument substituted for the result SQL parameter of *R*.

2) If *RV* is not the null value and the most specific type of *RV* is not compatible with *MAT*, then an exception condition is raised: *data exception — most specific type mismatch*.

ii) Let *ERDT* be the effective returns data type of the <routine invocation>.

iii) Let the result of the <routine invocation> be the result the application of the General Rules of Subclause 9.2, "Store assignment", with *RV* as *VALUE* and *ERDT* as *TARGET*.

b) Otherwise, for each SQL parameter P_i of *R* that is an output SQL parameter or both an input SQL parameter and an output SQL parameter, let TS_i be the <target specification> of the corresponding <SQL argument> A_i.

Case:

i) If TS_i is a <host parameter specification> or an <embedded variable specification>, then the General Rules of Subclause 9.1, "Retrieval assignment", are applied with CPV_i as *VALUE* and TS_i as *TARGET*.

ii) If TS_i is an <SQL parameter reference>, a <column reference>, or a <target array element specification>, then

NOTE 245 — The <column reference> can only be a new transition variable column reference.

Case:

1) If <target array element specification> is specified, then

Case:

A) If the value of TS_i, denoted by *C*, is the null value, then an exception condition is raised: *data exception — null value in array target*.

B) Otherwise:

I) Let *N* be the maximum cardinality of *C*.

II) Let *M* be the cardinality of the value of *C*.

III) Let *I* be the value of the <simple value specification> immediately contained in TS_i.

IV) Let *EDT* be the element type of *C*.

V) Case:

1) If *I* is greater than zero and less than or equal to *M*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *M* derived as follows:

a) For *j* varying from 1 (one) to *I*–1 and from *I*+1 to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

b) The *I*-th element of *A* is set to the value of CPV_i , denoted by *SV*, by applying the General Rules of Subclause 9.2, "Store assignment", to the *I*-th element of *A* and *SV* as *TARGET* and *VALUE*, respectively.

2) If *I* is greater than *M* and less than or equal to *N*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *I* derived as follows:

a) For *j* varying from 1 (one) to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

b) For j varying from M+1 to I, the j-th element in A is the null value.

c) The I-th element of A is set to the value of CPV_i, denoted by SV, by applying the General Rules of Subclause 9.2, "Store assignment", to the I-th element of A and SV as *TARGET* and *VALUE*, respectively.

3) Otherwise, an exception condition is raised: *data exception — array element error*.

2) Otherwise, the General Rules of Subclause 9.2, "Store assignment" are applied with CPV_i as *VALUE* and TS_i as *TARGET*.

10) If *RSS* is not empty, then let *PR* be the descriptor of *SIP*.

a) Let *MAX* be the maximum number of returned result sets included in *PR*.

b) Let *OPN* be the actual number of returned result sets included in *RSS*.

c) Case:

i) If *OPN* is greater than *MAX*, then:

1) Let *RTN* be *MAX*.

2) A completion condition is raised: *warning — attempt to return too many result sets*.

ii) Otherwise, let *RTN* be *OPN*.

d) For each i, 1 (one) $\leq i \leq RTN$, let FRC_i be the with-return cursor of the i-th returned result set RS_i in *RSS*.

e) For each i, 1 (one) $\leq i \leq RTN$,

Case:

i) If FRC_i is a scrollable cursor, then the initial cursor position of RS_i is the current cursor position of FRC_i.

ii) Otherwise,

Case:

1) If the application of Subclause 15.3, "Determination of the current row of a cursor", with FRC_i as *CURSOR* and NEXT as *FETCH ORIENTATION*, would position the cursor on or before some row in RS_i, then let *RN* be the ordinal position of that row in RS_i.

2) Otherwise, let *RN* be one greater than the number of rows in RS_i.

The first *RN* rows are deleted from RS_i and the initial cursor position of RS_i is before the first row.

f) A completion condition is raised: *warning — result sets returned*.

11) If *R* is an SQL routine, then for every open cursor *CR* that was opened during the execution of *R* and that is either a declared cursor or a local extended dynamic cursor, the General Rules of Subclause 15.4, "Effect of closing a cursor", are applied, with *CR* as *CURSOR* and SAVE as *DISPOSITION*.

12) If *R* is an external routine, then it is implementation-defined whether, for every open cursor *CR* that was opened during the execution of *R* and that is either a declared cursor or a local extended dynamic cursor:

    a) The the General Rules of Subclause 15.4, “Effect of closing a cursor”, are applied, with *CR* as *CURSOR* and SAVE as *DISPOSITION*.

    b) The cursor instance descriptor of *CR* is destroyed.

13) Prepare *CSC* to become the current SQL-session context:

    a) Set the value of the current constraint mode for each integrity constraint in *CSC* to the value of the current constraint mode for each integrity constraint in *RSC*.

    b) Set the value of the current access mode in *CSC* to the value of the current access mode in *RSC*.

    c) Set the value of the current isolation level in *CSC* to the value of the current isolation level in *RSC*.

    d) Set the value of the current condition area limit in *CSC* to the value of the current condition area limit *CAL* in *RSC*.

    e) For each occupied condition area *CA* in the first diagnostics area of *RSC*, if the value of RETURNED_SQLSTATE in *CA* does not represent *successful completion*, then

       Case:

       i) If the number of occupied condition areas in the first diagnostics area *DA1* in *CSC* is less than *CAL*, then *CA* is copied to the first vacant condition area in *DA1*.

          NOTE 246 — This causes the first vacant condition area in *DA1* to become occupied.

       ii) Otherwise, the value of MORE in the statement information area of *DA1* is set to 'Y'.

    f) Replace the identities of all instances of global temporary tables in *CSC* with the identities of the instances of global temporary tables in *RSC*.

    g) Remove the top cell from the authorization stack of *RSC* and set the authorization stack of *CSC* to a copy of the authorization stack of *RSC*.

       NOTE 247 — The copying of *RSC*'s authorization stack into *CSC* is necessary in order to carry back any change in the SQL-session user identifier.

    h) If the subject routine is an SQL-invoked procedure, then the result set sequence *RSS* is added to *CSC*.

       NOTE 248 — *RSS* is now available for reference by an <allocate cursor statement> containing FOR PROCEDURE.

14) If *R* is an SQL-invoked function or if *R* is an SQL-invoked procedure and the descriptor of *R* includes an indication that a new savepoint level is to be established when *R* is invoked, then the current savepoint level is destroyed.

15) *CSC* becomes the current SQL-session context.

Conformance Rules

1) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <generalized expression>.

2) Without Feature S201, “SQL-invoked routines on arrays”, conforming SQL language shall not contain an <SQL argument> whose declared type is an array type.

3) Without Feature S202, “SQL-invoked routines on multisets”, conforming SQL language shall not contain an <SQL argument> whose declared type is a multiset type.

4) Without Feature B033, “Untyped SQL-invoked function arguments”, conforming SQL language shall not contain a <routine invocation> that is not simply contained in a <call statement> that simply contains an <SQL argument> that is a <dynamic parameter specification>.

10.5 <character set specification>

Function

Identify a character set.

Format

```
<character set specification> ::=
    <standard character set name>
  | <implementation-defined character set name>
  | <user-defined character set name>

<standard character set name> ::=
  <character set name>

<implementation-defined character set name> ::=
  <character set name>

<user-defined character set name> ::=
  <character set name>
```

Syntax Rules

1) The <standard character set name>s and <implementation-defined character set name>s that are supported are implementation-defined.

2) A character set identified by a <standard character set name>, or by an <implementation-defined character set name> has associated with it a privilege descriptor that was effectively defined by the <grant statement>

   ```
   GRANT USAGE ON
   CHARACTER SET CS
   TO PUBLIC
   ```

   where *CS* is the <character set name> contained in the <character set specification>. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

3) The <standard character set name>s shall include SQL_CHARACTER and those character sets specified in Subclause 4.2.7, "Character sets", as defined by this and other standards.

4) The <implementation-defined character set name>s shall include SQL_TEXT and SQL_IDENTIFIER.

5) Let *C* be the <character set name> contained in the <character set specification>. The schema identified by the explicit or implicit qualifier of the <character set name> shall include the descriptor of *C*.

6) If a <character set specification> is not contained in a <schema definition>, then the <character set name> immediately contained in the <character set definition> shall contain an explicit <schema name> that is not equivalent to INFORMATION_SCHEMA.

Access Rules

1) Case:

a) If <character set specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on *C*.

b) Otherwise, the current privileges shall include USAGE on *C*.

General Rules

1) A <character set specification> identifies a character set. Let the identified character set be *CS*.

2) A <standard character set name> specifies the name of a character set that is defined by a national or international standard. The character repertoire of *CS* is defined by the standard defining the character set identified by that <standard character set name>. The default collation of the character set is defined by the order of the characters in the standard and has the PAD SPACE characteristic.

3) An <implementation-defined character set name> specifies the name of a character set that is implementation-defined. The character repertoire of *CS* is implementation-defined. The default collation of the character set and whether the collation has the NO PAD characteristic or the PAD SPACE characteristic is implementation-defined.

4) A <user-defined character set name> identifies a character set whose descriptor is included in some schema whose <schema name> is not equivalent to INFORMATION_SCHEMA.

   NOTE 249 — The default collation of the character set is defined as in Subclause 11.31, "<character set definition>".

5) There is a character set descriptor for every character set that can be specified by a <character set specification>.

Conformance Rules

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

10.6 <specific routine designator>

Function

Specify an SQL-invoked routine.

Format

```
<specific routine designator> ::=
    SPECIFIC <routine type> <specific name>
  | <routine type> <member name> [ FOR <schema-resolved user-defined type name> ]

<routine type> ::=
    ROUTINE
  | FUNCTION
  | PROCEDURE
  | [ INSTANCE | STATIC | CONSTRUCTOR ] METHOD

<member name> ::=
  <member name alternatives> [ <data type list> ]

<member name alternatives> ::=
    <schema qualified routine name>
  | <method name>

<data type list> ::=
  <left paren> [ <data type> [ { <comma> <data type> }... ] ] <right paren>
```

Syntax Rules

1) If a <specific name> *SN* is specified, then the <specific routine designator> shall identify an SQL-invoked routine whose <specific name> is *SN*.

2) If <routine type> specifies METHOD and none of INSTANCE, STATIC, or CONSTRUCTOR is specified, then INSTANCE is implicit.

3) If a <member name> *MN* is specified, then:

   a) If <schema-resolved user-defined type name> is specified, then <routine type> shall specify METHOD. If METHOD is specified, then <schema-resolved user-defined type name> shall be specified.

   b) Case:

      i) If <routine type> specifies METHOD, then <method name> shall be specified. Let *SCN* be the implicit or explicit <schema name> of <schema-resolved user-defined type name>, let *METH* be the <method name>, and let *RN* be `SCN.METH`.

      ii) Otherwise, <schema qualified routine name> shall be specified. Let *RN* be the <schema qualified routine name> of *MN* and let *SCN* be the <schema name> of *MN*.

   c) Case:

      i) If *MN* contains a <data type list>, then:

1) If <routine type> specifies FUNCTION, then there shall be exactly one SQL-invoked regular function in the schema identified by *SCN* whose <schema qualified routine name> is *RN* such that for all *i* the Syntax Rules of Subclause 9.18, "Data type identity", when applied with the declared type of its *i*-th SQL parameter and the *i*-th <data type> in the <data type list> of *MN*, are satisfied. The <specific routine designator> identifies that SQL-invoked function.

2) If <routine type> specifies PROCEDURE, then there shall be exactly one SQL-invoked procedure in the schema identified by *SCN* whose <schema qualified routine name> is *RN* such that for all *i* the Syntax Rules of Subclause 9.18, "Data type identity", when applied with the declared type of its *i*-th SQL parameter and the *i*-th <data type> in the <data type list> of *MN*, are satisfied. The <specific routine designator> identifies that SQL-invoked procedure.

3) If <routine type> specifies METHOD, then

   Case:

   A) If STATIC is specified, then there shall be exactly one static SQL-invoked method of the type identified by <schema-resolved user-defined type name> whose <method name> is *METH*, such that for all *i* the Syntax Rules of Subclause 9.18, "Data type identity", when applied with the declared data type of its *i*-th SQL parameter and the *i*-th <data type> in the <data type list> of *MN*, are satisfied. The <specific routine designator> identifies that static SQL-invoked method.

   B) If CONSTRUCTOR is specified, then there shall be exactly one SQL-invoked constructor method of the type identified by <schema-resolved user-defined type name> whose <method name> is *METH*, such that for all *i* the Syntax Rules of Subclause 9.18, "Data type identity", when applied with the declared data type of its *i*-th SQL parameter in the unaugmented <SQL parameter declaration list> amd the *i*-th <data type> in the <data type list> of *MN*, are satisfied. The <specific routine designator> identifies that SQL-invoked constructor method.

   C) Otherwise, there shall be exactly one instance SQL-invoked method of the type identified by <schema-resolved user-defined type name> whose <method name> is *METH*, such that for all *i* the Syntax Rules of Subclause 9.18, "Data type identity", when applied with the declared data type of its *i*-th SQL parameter in the unaugmented <SQL parameter declaration list> and the *i*-th <data type> in the <data type list> of *MN*, are satisfied. The <specific routine designator> identifies that instance SQL-invoked method.

4) If <routine type> specifies ROUTINE, then there shall be exactly one SQL-invoked routine in the schema identified by *SCN* whose <schema qualified routine name> is *RN* such that for all *i* the Syntax Rules of Subclause 9.18, "Data type identity", when applied with the declared type of its *i*-th SQL parameter and the *i*-th <data type> in the <data type list> of *MN*, are satisfied. The <specific routine designator> identifies that SQL-invoked routine.

ii) Otherwise:

1) If <routine type> specifies FUNCTION, then there shall be exactly one SQL-invoked function in the schema identified by *SCN* whose <schema qualified routine name> is *RN*. The <specific routine designator> identifies that SQL-invoked function.

2) If <routine type> specifies PROCEDURE, then there shall be exactly one SQL-invoked procedure in the schema identified by *SCN* whose <schema qualified routine name> is *RN*. The <specific routine designator> identifies that SQL-invoked procedure.

3) If <routine type> specifies METHOD, then

   Case:

   A) If STATIC is specified, then there shall be exactly one static SQL-invoked method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH*. The <specific routine designator> identifies that static SQL-invoked method.

   B) If CONSTRUCTOR is specified, then there shall be exactly one SQL-invoked constructor method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH*. The <specific routine designator> identifies that SQL-invoked constructor method.

   C) Otherwise, there shall be exactly one instance SQL-invoked method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH*. The <specific routine designator> identifies that instance SQL-invoked method.

4) If <routine type> specifies ROUTINE, then there shall be exactly one SQL-invoked routine in the schema identified by *SCN* whose <schema qualified routine name> is *RN*. The <specific routine designator> identifies that SQL-invoked routine.

4) If FUNCTION is specified, then the SQL-invoked routine that is identified shall be an SQL-invoked regular function. If PROCEDURE is specified, then the SQL-invoked routine that is identified shall be an SQL-invoked procedure. If STATIC METHOD is specified, then the SQL-invoked routine that is identified shall be a static SQL-invoked method. If CONSTRUCTOR METHOD is specified, then the SQL-invoked routine shall be an SQL-invoked constructor method. If INSTANCE METHOD is specified or implicit, then the SQL-invoked routine shall be an instance SQL-invoked method. If ROUTINE is specified, then the SQL-invoked routine that is identified is either an SQL-invoked function or an SQL-invoked procedure.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <specific routine designator> that contains a <routine type> that immediately contains METHOD.

10.7 <collate clause>

Function

Specify a default collation.

Format

```
<collate clause> ::=
  COLLATE <collation name>
```

Syntax Rules

1) Let *C* be the <collation name> contained in the <collate clause>. The schema identified by the explicit or implicit qualifier of the <collation name> shall include the descriptor of *C*.

Access Rules

1) Case:

   a) If <collate clause> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on *C*.

   b) Otherwise, the current privileges shall include USAGE on *C*.

General Rules

None.

Conformance Rules

1) Without Feature F690, “Collation support”, conforming SQL language shall not contain a <collate clause>.

10.8 <constraint name definition> and <constraint characteristics>

Function

Specify the name of a constraint and its characteristics.

Format

```
<constraint name definition> ::=
  CONSTRAINT <constraint name>

<constraint characteristics> ::=
    <constraint check time> [ [ NOT ] DEFERRABLE ]
  | [ NOT ] DEFERRABLE [ <constraint check time> ]

<constraint check time> ::=
    INITIALLY DEFERRED
  | INITIALLY IMMEDIATE
```

Syntax Rules

1) If <constraint check time> is not specified, then INITIALLY IMMEDIATE is implicit.

2) Case:

   a) If INITIALLY DEFERRED is specified, then:

      i) NOT DEFERRABLE shall not be specified.

      ii) If DEFERRABLE is not specified, then DEFERRABLE is implicit.

   b) If INITIALLY IMMEDIATE is specified or implicit and neither DEFERRABLE nor NOT DEFERRABLE is specified, then NOT DEFERRABLE is implicit.

Access Rules

None.

General Rules

1) Let *C* be the constraint identified by <constraint name>.

2) If NOT DEFERRABLE is specified, then *C* is not deferrable; otherwise it is deferrable.

3) If <constraint check time> is INITIALLY DEFERRED, then the initial constraint mode for *C* is *deferred*; otherwise, the initial constraint mode for *C* is *immediate*.

Conformance Rules

1) Without Feature F721, “Deferrable constraints”, conforming SQL language shall not contain a <constraint characteristics>.

   NOTE 250 — This means that INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

2) Without Feature F491, “Constraint management”, conforming SQL language shall not contain a <constraint name definition>.

10.9 <aggregate function>

This Subclause is modified by Subclause 11.2, "<aggregate function>", in ISO/IEC 9075-14.

Function

Specify a value computed from a collection of rows.

Format

```
<aggregate function> ::=
    COUNT <left paren> <asterisk> <right paren> [ <filter clause> ]
  | <general set function> [ <filter clause> ]
  | <binary set function> [ <filter clause> ]
  | <ordered set function> [ <filter clause> ]

<general set function> ::=
  <set function type> <left paren> [ <set quantifier> ]
      <value expression> <right paren>

<set function type> ::=
  <computational operation>

<computational operation> ::=
    AVG
  | MAX
  | MIN
  | SUM
  | EVERY
  | ANY
  | SOME
  | COUNT
  | STDDEV_POP
  | STDDEV_SAMP
  | VAR_SAMP
  | VAR_POP
  | COLLECT
  | FUSION
  | INTERSECTION

<set quantifier> ::=
    DISTINCT
  | ALL

<filter clause> ::=
  FILTER <left paren> WHERE <search condition> <right paren>

<binary set function> ::=
  <binary set function type> <left paren> <dependent variable expression> <comma>
      <independent variable expression> <right paren>

<binary set function type> ::=
    COVAR_POP
  | COVAR_SAMP
  | CORR
```

```
  | REGR_SLOPE
  | REGR_INTERCEPT
  | REGR_COUNT
  | REGR_R2
  | REGR_AVGX
  | REGR_AVGY
  | REGR_SXX
  | REGR_SYY
  | REGR_SXY

<dependent variable expression> ::=
  <numeric value expression>

<independent variable expression> ::=
  <numeric value expression>

<ordered set function> ::=
    <hypothetical set function>
  | <inverse distribution function>

<hypothetical set function> ::=
  <rank function type> <left paren>
      <hypothetical set function value expression list> <right paren>
      <within group specification>

<within group specification> ::=
  WITHIN GROUP <left paren> ORDER BY <sort specification list> <right paren>

<hypothetical set function value expression list> ::=
  <value expression> [ { <comma> <value expression> }... ]

<inverse distribution function> ::=
  <inverse distribution function type> <left paren>
      <inverse distribution function argument> <right paren>
      <within group specification>

<inverse distribution function argument> ::=
  <numeric value expression>

<inverse distribution function type> ::=
    PERCENTILE_CONT
  | PERCENTILE_DISC
```

Syntax Rules

1) Let *AF* be the <aggregate function>.

2) If STDDEV_POP, STDDEV_SAMP, VAR_POP, or VAR_SAMP is specified, then <set quantifier> shall not be specified.

3) If <general set function> is specified and <set quantifier> is not specified, then ALL is implicit.

4) The argument source of an <aggregate function> is

   Case:

   a) If *AF* is immediately contained in a <set function specification>, then a table or group of a grouped table as specified in Subclause 7.10, “<having clause>”, and Subclause 7.12, “<query specification>”.

b) Otherwise, the collection of rows in the current row's window frame defined by the window structure descriptor identified by the <window function> that simply contains *AF*, as defined in Subclause 7.11, "<window clause>".

5) Let *T* be the argument source of *AF*.

6) If COUNT is specified, then the declared type of the result is an implementation-defined exact numeric type scale of 0 (zero).

7) If <general set function> is specified, then:

a) The <value expression> *VE* shall not contain a <window function>.

b) Let *DT* be the declared type of the <value expression>.

c) If *AF* specifies a <general set function> whose <set quantifier> is DISTINCT, then *VE* is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

d) If *AF* specifies a <set function type> that is MAX or MIN, then *VE* is an operand of an ordering operation. The Syntax Rules of Subclause 9.12, "Ordering operations", apply.

e) If EVERY, ANY, or SOME is specified, then *DT* shall be boolean and the declared type of the result is boolean.

f) If MAX or MIN is specified, then the declared type of the result is *DT*.

g) If SUM or AVG is specified, then:

i) *DT* shall be a numeric type or an interval type.

ii) If SUM is specified and *DT* is exact numeric with scale *S*, then the declared type of the result is an implementation-defined exact numeric type with scale *S*.

iii) If AVG is specified and *DT* is exact numeric, then the declared type of the result is an implementation-defined exact numeric type with precision not less than the precision of *DT* and scale not less than the scale of *DT*.

iv) If *DT* is approximate numeric, then the declared type of the result is an implementation-defined approximate numeric with precision not less than the precision of *DT*.

v) If *DT* is interval, then the declared type of the result is interval with the same precision as *DT*.

h) If VAR_POP or VAR_SAMP is specified, then the declared type of the result is an implementation-defined approximate numeric type. If *DT* is an approximate numeric type, then the precision of the result is not less than the precision of *DT*.

i) `STDDEV_POP(`*X*`)` is equivalent to `SQRT(VAR_POP(`*X*`))`.

j) `STDDEV_SAMP(`*X*`)` is equivalent to `SQRT(VAR_SAMP(`*X*`))`.

k) If COLLECT is specified, then the declared type of the result is *DT* MULTISET.

l) `COLLECT (`*X*`)` is equivalent to `FUSION (MULTISET [`*X*`])`.

m) If FUSION is specified, then *DT* shall be a multiset type, and DISTINCT shall not be specified. The declared type of the result is *DT*.

n) If INTERSECTION is specified, then *DT* shall be a multiset type, and DISTINCT shall not be specified. *VE* is a multiset operand of a multiset element grouping operation, and the Syntax Rules of Subclause 9.11, "Multiset element grouping operations", apply. The declared type of the result is *DT*.

8) A <filter clause> shall not contain a <query expression>, a <window function>, or an outer reference.

9) If <binary set function> is specified, then:

a) The <dependent variable expression> *DVE* and the <independent variable expression> *IVE* shall not contain a <window function>.

b) Let *DTDVE* be the declared type of *DVE* and let *DTIVE* be the declared type of *IVE*.

c) Case:

i) The declared type of REGR_COUNT is an implementation-defined exact numeric type with scale of 0 (zero).

ii) Otherwise, the declared type of the result is an implementation-defined approximate numeric type. If *DTDVE* is an approximate numeric type, then the precision of the result is not less than the precision of *DTDVE*. If *DTIVE* is an approximate numeric type, then the precision of the result is not less than the precision of *DTIVE*.

10) If <hypothetical set function> is specified, then:

a) The <hypothetical set function> shall not contain a <window function>, a <set function specification>, or a <query expression>.

b) The number of <value expression>s simply contained in <hypothetical set function value expression list> shall be the same as the number of <sort key>s simply contained in the <sort specification list>.

c) For each <value expression> *HSFVE* simply contained in the <hypothetical set function value expression list>, let *SK* be the corresponding <sort key> simply contained in the <sort specification list>.

Case:

i) If the declared type of *HSFVE* is a character string type, then the declared type of *SK* shall be a character string type with the same character repertoire as that of *HSFVE*. The collation is determined by applying Subclause 9.13, "Collation determination", with operands *HSFVE* and *SK*.

ii) Otherwise the declared types of *HSFVE* and *SK* shall be compatible.

d) Case:

i) If RANK or DENSE_RANK is specified, then the declared type of the result is exact numeric with implementation-defined precision and with scale 0 (zero).

ii) Otherwise, the declared type of the result is approximate numeric with implemenation-defined precision.

11) If <inverse distribution function> is specified, then:

a) The <within group specification> shall contain a single <sort specification>.

b) The <inverse distribution function> shall not contain a <window function>, a <set function specification>, or a <query expression>.

c) Let *DT* be the declared type of the <value expression> simply contained in the <sort specification>.

d) If PERCENTILE_CONT is specified, then *DT* shall be numeric or interval.

e) The declared type of the result is

Case:

i) If *DT* is numeric, then approximate numeric with implementation-defined precision.

ii) If *DT* is interval, then *DT*.

Access Rules

None.

General Rules

1) If, during the computation of the result of *AF*, an intermediate result is not representable in the declared type of the site that contains that intermediate result, then

Case:

a) If the most specific type of the result of *AF* is an interval type, then an exception condition is raised: *data exception — interval value out of range*.

b) If the most specific type of the result of *AF* is a multiset type, then an exception condition is raised: *data exception — multiset value overflow*.

c) Otherwise, an exception condition is raised: *data exception — numeric value out of range*.

2) Case:

a) If <filter clause> is specified, then the <search condition> is effectively evaluated for each row of *T*. Let *T1* be the collection of rows of *T* for which the result of the <search condition> is *True*.

b) Otherwise, let *T1* be *T*.

3) If COUNT(*) is specified, then the result is the cardinality of *T1*.

4) If <general set function> is specified, then:

a) Let *TX* be the single-column table that is the result of applying the <value expression> to each row of *T1* and eliminating null values. If one or more null values are eliminated, then a completion condition is raised: *warning — null value eliminated in set function*.

b) Case:

i) If DISTINCT is specified, then let *TXA* be the result of eliminating redundant duplicate values from *TX*, using the comparison rules specified in Subclause 8.2, "<comparison predicate>", to identify the redundant duplicate values.

ii) Otherwise, let *TXA* be *TX*.

c) Let *N* be the cardinality of *TXA*.

d) Case:

i) If COUNT is specified, then the result is *N*.

ii) If *TXA* is empty, then the result is the null value.

iii) If AVG is specified, then the result is the average of the values in *TXA*.

iv) If MAX or MIN is specified, then the result is respectively the maximum or minimum value in *TXA*. These results are determined using the comparison rules specified in Subclause 8.2, "<comparison predicate>". If *DT* is a user-defined type and the comparison of two values in *TXA* results in *Unknown*, then the maximum or minimum of *TXA* is implementation-dependent.

v) If SUM is specified, then the result is the sum of the values in *TXA*. If the sum is not within the range of the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range*.

vi) If EVERY is specified, then

Case:

1) If the value of some element of *TXA* is *False*, then the result is *False*.

2) Otherwise, the result is *True*.

vii) If ANY or SOME is specified, then

Case:

1) If the value of some element of *TXA* is *True*, then the result is *True*.

2) Otherwise, the result is *False*.

viii) If VAR_POP or VAR_SAMP is specified, then let *S1* be the sum of values in the column of *TXA*, and *S2* be the sum of the squares of the values in the column of *TXA*.

1) If VAR_POP is specified, then the result is `(S2-S1*S1/N)/N`.

2) If VAR_SAMP is specified, then

Case:

A) If *N* is 1 (one), then the result is the null value.

B) Otherwise, the result is `(S2-S1*S1/N)/(N-1)`

ix) If FUSION is specified, then the result is the multiset *M* such that for each value *V* in the element type of *DT*, including the null value, the number of elements of *M* that are identical to *V* is the sum of the number of identical copies of *V* in the multisets that are the values of the column in each row of *TXA*.

x) If INTERSECTION is specified, then the result is a multiset *M* such that for each value *V* in the element type of *DT*, including the null value, the number of duplicates of *V* in *M* is the minimum of the number of duplicates of *V* in the multisets that are the values of the column in each row of *TXA*.

NOTE 251 — This rule says "the result is a multiset" rather than "the result is the multiset" because the precise duplicate values are not specified. Thus this calculation is non-deterministic for certain element types, namely those based on character string, datetime with time zone and user-defined types.

NOTE 252 — The notion of one data type being based on another data type is defined in Subclause 4.1, "Data types".

5) If <binary set function type> is specified, then:

a) Let *TXA* be the two-column table that is the result of applying the <dependent variable expression> and the <independent variable expression> to each row of *T1* and eliminating each row in which either <dependent variable expression> or <independent variable expression> is the null value. If one or more null values are eliminated, then a completion condition is raised: *warning — null value eliminated in set function*.

b) Let *N* be the cardinality of *TXA*, let *SUMX* be the sum of the column of values of <independent variable expression>, let *SUMY* be the sum of the column of values of <dependent variable expression>, let *SUMX2* be the sum of the squares of values in the <independent variable expression> column, let *SUMY2* be the sum of the squares of values in the <dependent variable expression> column, and let *SUMXY* be the sum of the row-wise products of the value in the <independent variable expression> column times the value in the <dependent variable expression> column.

c) Case:

i) If REGR_COUNT is specified, then the result is *N*.

ii) If *N* is 0 (zero), then the result is the null value.

iii) If REGR_SXX is specified, then the result is `(SUMX2-SUMX*SUMX/N)`.

iv) If REGR_SYY is specified, then the result is `(SUMY2-SUMY*SUMY/N)`.

v) If REGR_SXY is specified, then the result is `(SUMXY-SUMX*SUMY/N)`.

vi) If REGR_AVGX is specified, then the result is `SUMX/N`.

vii) If REGR_AVGY is specified, then the result is `SUMY/N`.

viii) If COVAR_POP is specified, then the result is `(SUMXY-SUMX*SUMY/N)/N`.

ix) If COVAR_SAMP is specified, then

Case:

1) If *N* is 1 (one), then the result is the null value.

2) Otherwise, the result is `(SUMXY-SUMX*SUMY/N)/(N-1)`

x) If CORR is specified, then

Case:

1) If *N***SUMX2* equals *SUMX***SUMX*, then the result is the null value.

NOTE 253 — In this case, all remaining values of <independent variable expression> are equal, and consequently the <independent variable expression> does not correlate with the <dependent variable expression>.

2) If *N***SUMY2* equals *SUMY***SUMY*, then the result is the null value.

NOTE 254 — In this case, all remaining values of <dependent variable expression> are equal, and consequently the <dependent variable expression> does not correlate with the <independent variable expression>.

3) Otherwise, the result is `SQRT(POWER(N*SUMXY-SUMX*SUMY,2) / ((N*SUMX2-SUMX*SUMX)*(N*SUMY2-SUMY*SUMY)))`. If the exponent of the

approximate mathematical result of the operation is not within the implementation-defined exponent range for the result data type, then the result is the null value.

xi) If REGR_R2 is specified, then

Case:

1) If *N***SUMX2* equals *SUMX***SUMX*, then the result is the null value.

NOTE 255 — In this case, all remaining values of <independent variable expression> are equal, and consequently the least-squares fit line would be vertical, or, if *N* = 1 (one), there is no uniquely determined least-squares-fit line.

2) If *N***SUMY2* equals *SUMY***SUMY*, then the result is 1 (one).

NOTE 256 — In this case, all remaining values of <dependent variable expression> are equal, and consequently the least-squares fit line is horizontal.

3) Otherwise, the result is `POWER(N*SUMXY-SUMX*SUMY,2) / ((N*SUMX2-SUMX*SUMX) * (N*SUMY2-SUMY*SUMY))`. If the exponent of the approximate mathematical result of the operation is not within the implementation-defined exponent range for the result data type, then the result is the null value.

xii) If REGR_SLOPE(*Y*, *X*) is specified, then

Case:

1) If *N***SUMX2* equals *SUMX***SUMX*, then the result is the null value.

NOTE 257 — In this case, all remaining values of <independent variable expression> are equal, and consequently the least-squares fit line would be vertical, or, if *N* = 1 (one), then there is no uniquely determined least-squares-fit line.

2) Otherwise, the result is `(N*SUMXY-SUMX*SUMY) / (N*SUMX2-SUMX*SUMX)`. If the exponent of the approximate mathematical result of the operation is not within the implementation-defined exponent range for the result data type, then the result is the null value.

xiii) If REGR_INTERCEPT is specified, then

Case:

1) If *N***SUMX2* equals *SUMX***SUMX*, then the result is the null value.

NOTE 258 — In this caes, all remaining values of <independent variable expression> are equal, and consequently the least-squares fit line would be vertical, or, if *N* = 1 (one), then there is no uniquely determined least-squares-fit line.

2) Otherwise, the result is `(SUMY*SUMX2-SUMX*SUMXY) / (N*SUMX2-SUMX*SUMX)`. If the exponent of the approximate mathematical result of the operation is not within the implementation-defined exponent range for the result data type, then the result is the null value.

6) If <hypothetical set function> is specified, then

a) Let *WIFT* be the <rank function type>.

b) Let *TNAME* be an implementation-dependent name for *T1*.

c) Let *K* be the number of <value expression>s simply contained in <hypothetical set function value expression list>.

d) Let VE_1, ..., VE_K be the <value expression>s simply contained in the <hypothetical set function value expression list>.

e) Let *WIFTVAL*, *MARKER* and CN_1, ..., CN_K be distinct implementation-dependent column names.

f) Let SP_1, ..., SP_K be the <sort specification>s simply contained in the <sort specification list>. For each *i*, let WSP_i be the <sort specification> obtained from SP_i by replacing the <sort key> with CN_i.

g) The result is the result of the <scalar subquery>

```
( SELECT WIFTVAL
 FROM ( SELECT MARKER, WIFT() OVER
           ( ORDER BY WSP1, ..., WSPK )
        FROM ( SELECT 0, SK1, ..., SKK
               FROM TNAME
                 UNION ALL
                    VALUES (1, VE1, ..., VEK) )
                 AS TXNAME (MARKER, CN1, ..., CNK )
      ) AS TEMPTABLE (MARKER, WIFTVAL)
 WHERE MARKER = 1 )
```

7) If <inverse distribution function> is specified, then

a) Let *NVE* be the value of the <inverse distribution function argument>.

b) If *NVE* is the null value, then the result is the null value.

c) If *NVE* is less than 0 (zero) or greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range*.

d) Let *TXA* be the single-column table that is the result of applying the <value expression> simply contained in the <sort specification> to each row of *T1* and eliminating null values. If one or more null values are eliminated, then a completion condition is raised: *warning — null value eliminated in set function*. *TXA* is ordered by the <sort specification> as specified in the General Rules of Subclause 10.10, "<sort specification list>".

e) Let *TXANAME* be an implementation-dependent name for *TXA*.

f) Let *TXCOLNAME* be an implementation-dependent column name for the column of *TXA*.

g) Let *WSP* be obtained from the <sort specification> by replacing the <sort key> with *TXCOLNAME*.

h) Case:

i) If PERCENTILE_CONT is specified, then:

1) Let *ROW0* be the greatest exact numeric value with scale 0 (zero) that is less than or equal to *NVE**(*N*–1). Let *ROWLIT0* be a <literal> representing *ROW0*.

2) Let *ROW1* be the least exact numeric value with scale 0 (zero) that is greater than or equal to *NVE**(*N*–1). Let *ROWLIT1* be a <literal> representing *ROW1*.

3) Let *FACTOR* be an <approximate numeric literal> representing *NVE**(*N*–1)–*ROW0*.

4) The result is the result of the <scalar subquery>

```
( WITH TEMPTABLE(X, Y) AS
```

```
        ( SELECT ROW_NUMBER()
                   OVER (ORDER BY WSP) - 1,
                 TXCOLNAME
          FROM TXANAME )
SELECT CAST ( T0.Y + FACTOR * (T1.Y - T0.Y) AS DT )
FROM TEMPTABLE T0, TEMPTABLE T1
WHERE T0.ROWNUMBER = ROWLIT0
  AND T1.ROWNUMBER = ROWLIT1 )
```

NOTE 259 — Although ROW_NUMBER is nondeterministic, the values of T0.Y and T1.Y are determined by this expression. Note that the only column of *TXA* is completely ordered by *WSP*. If *NVE**(*N*–1) is a whole number, then the rows selected from T0 and T1 are the same and the result is just T0.Y. Otherwise, the subquery performs a linear interpolation between the two consecutive values whose row numbers in the ordered set, seen as proportions of the whole, bound the argument of the PERCENTILE_CONT operator.

ii) If PERCENTILE_DISC is specified, then

1) If the <ordering specification> simply contained in *WSP* is DESC, then let *MAXORMIN* be MAX; otherwise let *MAXORMIN* be MIN.

2) Let *NVELIT* be a <literal> representing the value of *NVE*.

3) The result is the result of the <scalar subquery>

```
( SELECT MAXORMIN (TXCOLNAME)
 FROM ( SELECT TXCOLNAME,
               CUME_DIST() OVER (ORDER BY WSP)
 FROM TXANAME ) AS TEMPTABLE (TXCOLNAME, CUMEDIST)
 WHERE CUMEDIST >= NVELIT )
```

Conformance Rules

1) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <computational operation> that immediately contains EVERY, ANY, or SOME.

2) Without Feature F561, “Full value expressions”, or Feature F801, “Full set function”, conforming SQL language shall not contain a <general set function> that immediately contains DISTINCT and contains a <value expression> that is not a column reference.

3) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <general set function> that contains a <computational operation> that immediately contains COUNT and does not contain a <set quantifier> that immediately contains DISTINCT.

4) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <general set function> that does not contain a <set quantifier> that immediately contains DISTINCT and that contains a <value expression> that contains a column reference that does not reference a column of *T*.

5) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <binary set function> that does not contain either a <dependent variable expression> or an <independent variable expression> that contains a column reference that references a column of *T*.

6) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <value expression> simply contained in a <general set function> that contains a column reference that is an outer reference where the <value expression> is not a column reference.

7) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <numeric value expression> simply contained in a <dependent variable expression> or an <independent variable expression> that contains a column reference that is an outer reference and in which the <numeric value expression> is not a column reference.

8) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a column reference contained in an <aggregate function> that contains a reference to a column derived from a <value expression> that generally contains an <aggregate function> *SFS2* without an intervening <routine invocation>.

9) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <computational operation> that immediately contains STDDEV_POP, STDDEV_SAMP, VAR_POP, or VAR_SAMP.

10) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <binary set function type>.

11) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <hypothetical set function> or an <inverse distribution function>.

12) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <filter clause>.

13) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <computational operation> that immediately contains COLLECT.

14) Without Feature S275, “Advanced multiset support”, conforming SQL language shall not contain a <computational operation> that immediately contains FUSION or INTERSECTION.

    NOTE 260 — If INTERSECTION is specified, then the Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

15) Without Feature T052, “MAX and MIN for row types”, conforming SQL language shall not contain a <computational operation> that immediately contains MAX or MIN in which the declared type of the <value expression> is a row type.

    NOTE 261 — If DISTINCT is specified, then the Conformance Rules of Subclause 9.10, “Grouping operations”, also apply. If MAX or MIN is specified, then the Conformance Rules of Subclause 9.12, “Ordering operations”, also apply.

16) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain a <hypothetical set function value expression list> or a <sort specification list> that simply contains a <value expression> that contains more than one column reference, one of which is an outer reference.

17) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain an <inverse distribution function> that contains an <inverse distribution function argument> or a <sort specification> that contains more than one column reference, one of which is an outer reference.

18) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain an <aggregate function> that contains a <general set function> whose simply contained <value expression> contains more than one column reference, one of which is an outer reference.

19) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain an <aggregate function> that contains a <binary set function> whose simply contained <dependent variable expression> or <independent variable expression> contains more than one column reference, one of which is an outer reference.

10.10 <sort specification list>

Function

Specify a sort order.

Format

```
<sort specification list> ::=
  <sort specification> [ { <comma> <sort specification> }... ]

<sort specification> ::=
  <sort key> [ <ordering specification> ] [ <null ordering> ]

<sort key> ::=
  <value expression>

<ordering specification> ::=
    ASC
  | DESC

<null ordering> ::=
    NULLS FIRST
  | NULLS LAST
```

Syntax Rules

1) Let *DT* be the declared type of the <value expression> simply contained in the <sort key> contained in a <sort specification>.

2) Each <value expression> simply contained in the <sort key> contained in a <sort specification> is an operand of an ordering operation. The Syntax Rules of Subclause 9.12, "Ordering operations", apply.

3) If <null ordering> is not specified, then an implementation-defined <null ordering> is implicit. The implementation-defined default for <null ordering> shall not depend on the context outside of <sort specification list>.

Access Rules

None.

General Rules

1) A <sort specification list> defines an ordering of rows, as follows:

   a) Let *N* be the number of <sort specification>s.

   b) Let K_i, 1 (one) $\leq i \leq N$, be the <sort key> contained in the *i*-th <sort specification>.

c) Each <sort specification> specifies the *sort direction* for the corresponding sort key K_i. If DESC is not specified in the *i*-th <sort specification>, then the sort direction for K_i is ascending and the applicable <comp op> is the <less than operator>. Otherwise, the sort direction for K_i is descending and the applicable <comp op> is the <greater than operator>.

d) Let *P* be any row of the collection of rows to be ordered, and let *Q* be any other row of the same collection of rows.

e) Let PV_i and QV_i be the values of K_i in *P* and *Q*, respectively. The relative position of rows *P* and *Q* in the result is determined by comparing PV_i and QV_i according to the rules of Subclause 8.2, "<comparison predicate>" where the <comp op> is the applicable <comp op> for K_i, with the following special treatment of null values.

Case:

i) If PV_i and QV_i are both the null value, then they are considered equal to each other.

ii) If PV_i is the null value and QV_i is not the null value, then

Case:

1) If NULLS FIRST is specified or implied, then PV_i <comp op> QV_i is considered to be *True*.

2) If NULLS LAST is specified or implied, then PV_i <comp op> QV_i is considered to be *False*.

iii) If PV_i is not the null value and QV_i is the null value, then

Case:

1) If NULLS FIRST is specified or implied, then PV_i <comp op> QV_i is considered to be *False*.

2) If NULLS LAST is specified or implied, then PV_i <comp op> QV_i is considered to be *True*.

f) PV_i is said to *precede* QV_i if the value of the <comparison predicate> "PV_i <comp op> QV_i" is *True* for the applicable <comp op>.

g) If PV_i and QV_i are not the null value and the result of "PV_i <comp op> QV_i" is *Unknown*, then the relative ordering of PV_i and QV_i is implementation-dependent.

h) The relative position of row *P* is before row *Q* if PV_n precedes QV_n for some *n*, 1 (one) ≤ *n* ≤ *N*, and PV_i is not distinct from QV_i for all *i* < *n*.

i) Two rows that are not distinct with respect to the <sort specification>s are said to be *peers* of each other. The relative ordering of peers is implementation-dependent.

Conformance Rules

1) Without Feature T611, "Elementary OLAP operations", conforming SQL language shall not contain a <null ordering>.

NOTE 262 — The Conformance Rules of Subclause 9.12, "Ordering operations", also apply.

10.11 Determination of view and view component privileges

Function

Determine view component privilege descriptors for all view components of a view, and the privilege descriptors of the view whose <action> is INSERT, UPDATE, or DELETE. In addition, determine the view privilege dependency descriptors of the view.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *V* be the *VIEW* in an application of this Subclause.

2) Let *A* be the <authorization identifier> that owns the schema identified by the <table name> of *V*.

3) Let VC_1, ..., VC_N be an enumeration of the view components of *V*. The enumeration shall have the following properties:

   a) For all *i* and *j* between 1 (one) and *N*, if VC_i is included in VC_j, then $i < j$.

   b) For all *i* and *j* between 1 (one) and *N*, if VC_i is a <query expression> simply contained in a <with list element> and VC_j references the table defined by VC_i, then $i < j$.

   NOTE 263 — A depth-first left-to-right traversal of the BNF of the <view definition> of *V* is one way to obtain such an enumeration.

4) If *V* effectively updatable, then the following subrules are performed to recursively create certain view component privilege descriptors. The following subrules also recursively define when a view component privilege is *immediately dependent* on another privilege descriptor.

   For each *i* between 1 (one) and *N*, in that order,

   Case:

   a) If VC_i is an updatable <query specification>, then:

      i) If the <from clause> VC_i contains exactly one <table reference> *TR*, then

         1) Case:

            A) If *TR* is a <table name> or <transition table name>, then let *S* be the set of applicable privileges for *A* on the table referenced by *TR*.

B) Otherwise, *TR* is a <query name> and references some VC_j, where $j < i$. Let *S* be the set of view component privilege descriptors whose identified object is VC_j or a column of VC_j.

2) If *S* contains a table privilege descriptor *PD* whose action is DELETE, then a view component table privilege descriptor is created as follows: the identified object is VC_i, the action is DELETE, the grantor is the special grantor value "_SYSTEM", and the grantee is *DELETE*. The privilege is grantable if and only if *PD* indicates a grantable privilege.

3) For each updatable column *C* of VC_i, let *CC* be the counterpart of *C* in the table identified by *TR*.

A) If *S* contains a column privilege descriptor *PD* whose action is UPDATE(*CC*), then a view component table privilege descriptor is created as follows: the identified object is *C*, the action is UPDATE, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if *PD* indicates a grantable privilege. The privilege descriptor is immediately dependent on *PD*.

B) If *V* is insertable-into, and *S* contains a column privilege descriptor *PD* whose action is INSERT(*CC*), then a view component table privilege descriptor is created as follows: the identified object is *C*, the action is INSERT, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if *PD* indicates a grantable privilege. The privilege descriptor is immediately dependent on *PD*.

ii) Otherwise:

1) If, for every leaf underlying table *LUT* of VC_i such that VC_i is one-to-one with *LUT*, the applicable privileges for *A* include DELETE on *LUT*, then a view component table privilege descriptor is created whose identified object is VC_i, action is DELETE, grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privileges for *A* includes grantable DELETE privilege on each such *LUT*. The privilege descriptor is immediately dependent on every privilege descriptor whose identified object is such a leaf underlying table, the action is DELETE, and grantee is *A*.

2) For each updatable column *C* of VC_i, let *LUT* be the leaf underlying table of VC_i that has a counterpart *CC* to *C*.

A) If the applicable privileges for *A* include UPDATE(*CC*) on *LUT*, then a view component column privilege descriptor *VCCPD* is created, as follows: the identified object is *C*, the action is UPDATE, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privilege for *A* includes grantable UPDATE(*CC*) privilege on *LUT*. The privilege descriptor is immediately dependent on every privilege descriptor whose identified object is *CC*, action is *UPDATE*, and grantee is *A*.

B) If *V* is insertable-into, and the applicable privileges for *A* include INSERT (CC) on LUT, then a view component column privilege descriptor *VCCPD* is created, as follows: the identified object is *C*, the action is INSERT, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privilege for *A* includes grantable INSERT(*CC*) privilege on *LUT*. The privilege descriptor is immediately dependent on every privilege descriptor whose identified object is *CC*, action is INSERT, and grantee is *A*.

b) If VC_i is a <table value constructor>, then there are no view component privilege descriptors that identify VC_i as object.

c) If VC_i is a <table value constructor>, then let *T* be the table identified by the <explicit table>.

i) If the applicable privileges for *A* include DELETE on *T*, then a view component table privilege descriptor is created whose identified object is VC_i, action is DELETE, grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privileges for *A* includes grantable DELETE privilege on *T*. The privilege descriptor is immediately dependent on the privilege descriptor whose identified object is *T*, action is DELETE, and grantee is *A*.

ii) For each updatable column *C* of VC_i, let *CC* be the counterpart to *C* in *T*. If the applicable privileges for *A* include UPDATE(*CC*) on *T*, then a view component column privilege descriptor *VCCPD* is created, as follows: the identified object is *C*, the action is UPDATE, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privilege for *A* includes grantable UPDATE(*CC*) privilege on *T*. The privilege descriptor is immediately dependent on the privilege descriptor whose identified object is *CC*, action is UPDATE, and grantee is *A*.

iii) For each updatable column *C* of VC_i, let *CC* be the counterpart to *C* in *T*. If the applicable privileges for *A* include INSERT(*CC*) on *T*, then a view component column privilege descriptor *VCCPD* is created, as follows: the identified object is *C*, the action is INSERT, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privilege for *A* includes grantable INSERT(*CC*) privilege on *T*. The privilege descriptor is immediately dependent on the privilege descriptor whose identified object is *CC*, action is INSERT, and grantee is *A*.

d) If $VC_i$is a <query expression>, then

Case:

i) If VC_i is a <simple table> *ST*, then there exists a $j < i$ such that *ST* is VC_j.

1) For each view component table privilege descriptor *VCTPD* of VC_j, a new view component table privilege descriptor is created, as follows: the identified object is VC_i, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the action, and the indication of whether the privilege is grantable is the same as in *VCTPD*. The privilege descriptor is immediately dependent on *VCTPD*.

2) For each view component column privilege descriptor *VCCPD* of VC_j, a new view component column privilege descriptor of VC_i is created, as follows: the identified object is the column of VC_i that is the counterpart of the column of VC_j identified by *VCCPD*, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the indication of whether the privilege is grantable is the same as in *VCCPD*. The privilege descriptor is immediately dependent on *VCTPD*.

ii) Otherwise, VC_i immediately contains UNION ALL. Let VC_l and VC_r be the left and right operands of VC_i, respectively.

1) If there is a view component table privilege descriptor $VCTPD_l$ whose identified object is VC_l and whose action is DELETE, and there is a view component table privilege descriptor

$VCTPD_r$ whose identified object is VC_r and whose action is DELETE, then a new view component table privilege descriptor is created as follows: the identified object is VC_i, the action is DELETE, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the privilege is grantable if and only if both $VCTPD_l$ and $VCTPD_r$ indicate that the privilege is grantable. The privilege descriptor is immediately dependent on $VCTPD_l$ and $VCTPD_r$.

2) For each updatable column *C* of VC_i, let C_l and C_r be the counterparts of *C* in VC_l and VC_r, respectively. If there is a view component column privilege descriptor $VCTPD_l$ whose identified object is C_l and whose action is UPDATE, and there is a view component column privilege descriptor $VCTPD_r$ whose identified object is C_r and whose action is UPDATE, then a new view component table privilege descriptor is created as follows: the identified object is *C*, the action is UPDATE, the grantor is the special grantor value "_SYSTEM" , the grantee is *A*, and the privilege is grantable if and only if both $VCTPD_l$ and $VCTPD_r$ indicate that the privilege is grantable. The privilege descriptor is immediately dependent on $VCTPD_l$ and $VCTPD_r$.

5) A view component privilege descriptor *SPD* is *simply dependent* on another privilege descriptor *PD* if *SPD* is immediately dependent on *PD*, or if there is a view component privilege descriptor *SPD2* such that *SPD* is immediately dependent on *SPD2* and *SPD2* is simply dependent on *PD*.

6) VC_N is the view component that is the <query expression> immediately contained in the <view definition>.

   a) For each view component table privilege descriptor *VCTPD* whose identified object is VC_N:

      i) A privilege descriptor *PD1* is created, as follows: the identified object is *V*, the action is the same as the action of *VCTPD*, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the privilege is grantable if and only if *VCTPD* is grantable.

      ii) For each privilege descriptor *PD* such that *VCTPD* is simply dependent on *PD*, and such that the object of *PD* is not a view component or a column of a view component, a view privilege dependency descriptor is created, as follows: the supporting privilege descriptor is *PD* and the dependent privilege descriptor is *PD1*.

   b) For each view component column privilege descriptor *VCCPD* whose identified object is VC_N:

      i) A privilege descriptor *PD2* is created, as follows: the identified object is the column of *V* that is the counterpart of the column identified by *VCCPD*, the action is the same as the action of *VCCPD*, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the privilege is grantable if and only if *VCTPD* is grantable.

      ii) For each privilege descriptor *PD* such that *VCCPD* is simply dependent on *PD*, and such that the object of *PD* is not a view component or a column of a view component, a view privilege dependency descriptor is created, as follows: the supporting privilege descriptor is *PD* and the dependent privilege descriptor is *PD2*.

7) If, for every column *C* of *V*, there is a column privilege descriptor *CPD* whose identified object is *C*, action is UPDATE, grantor is the special grantor value "_SYSTEM", and grantee is *A*, then it is implementation-defined whether a table privilege descriptor *TPD* is created whose identified object is *V*, action is UPDATE, grantor is the special grantor value "_SYSTEM", grantee is *A*.

If *TPD* is created, then a collection of view privilege dependency descriptors is created, one for each column *C* of *V*, in which the supporting privilege descriptor is *CPD* and the dependent privilege descriptor is *TPD*.

8) If, for every column *C* of *V*, there is a column privilege descriptor *CPD* whose identified object is *C*, action is INSERT, grantor is the special grantor value "_SYSTEM", grantee is *A*, then it is implementation-defined whether a table privilege descriptor *TPD* is created whose identified object is *V*, action is INSERT, grantor is the special grantor value "_SYSTEM", grantee is *A*. If such a table privilege descriptor is created, then it is directly dependent on every such column privilege descriptor, and it has an indication that the privilege is grantable if every such column privilege descriptor has an indication that it is grantable.

   If *TPD* is created, then a collection of view privilege dependency descriptors is created, one for each column *C* of *V*, in which the supporting privilege descriptor is *CPD* and the dependent privilege descriptor is *TPD*.

9) All view component privilege descriptors are destroyed.

Conformance Rules

None.

11 Schema definition and manipulation

This Clause is modified by Clause 9, "Schema definition and manipulation", in ISO/IEC 9075-4.
This Clause is modified by Clause 11, "Schema definition and manipulation", in ISO/IEC 9075-9.
This Clause is modified by Clause 9, "Schema definition and manipulation", in ISO/IEC 9075-13.
This Clause is modified by Clause 12, "Schema definition and manipulation", in ISO/IEC 9075-14.

11.1 <schema definition>

This Subclause is modified by Subclause 9.1, "<schema definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.1, "<schema definition>", in ISO/IEC 9075-9.

Function

Define a schema.

Format

```
<schema definition> ::=
  CREATE SCHEMA <schema name clause>
      [ <schema character set or path> ]
      [ <schema element>... ]

<schema character set or path> ::=
    <schema character set specification>
  | <schema path specification>
  | <schema character set specification> <schema path specification>
  | <schema path specification> <schema character set specification>

<schema name clause> ::=
    <schema name>
  | AUTHORIZATION <schema authorization identifier>
  | <schema name> AUTHORIZATION <schema authorization identifier>

<schema authorization identifier> ::=
  <authorization identifier>

<schema character set specification> ::=
  DEFAULT CHARACTER SET <character set specification>

<schema path specification> ::=
  <path specification>

<schema element> ::=
    <table definition>
  | <view definition>
  | <domain definition>
```

```
| <character set definition>
| <collation definition>
| <transliteration definition>
| <assertion definition>
| <trigger definition>
| <user-defined type definition>
| <user-defined cast definition>
| <user-defined ordering definition>
| <transform definition>
| <schema routine>
| <sequence generator definition>
| <grant statement>
| <role definition>
```

Syntax Rules

1) If <schema name> is not specified, then a <schema name> equal to <schema authorization identifier> is implicit.

2) If AUTHORIZATION <schema authorization identifier> is not specified, then

   Case:

   a) If the <schema definition> is contained in an SQL-client module that has a <module authorization identifier> specified, then an <authorization identifier> equal to that <module authorization identifier> is implicit for the <schema definition>.

   b) Otherwise, an <authorization identifier> equal to the SQL-session user identifier is implicit.

3) The <unqualified schema name> of the explicit or implicit <schema name> shall not be equivalent to the <unqualified schema name> of the <schema name> of any other schema in the catalog identified by the <catalog name> of <schema name>.

4) If a <schema definition> is contained in an <externally-invoked procedure> in an <SQL-client module definition>, then the effective <schema authorization identifier> and <schema name> during processing of the <schema definition> are, respectively, the <schema authorization identifier> and <schema name> specified or implicit in the <schema definition>.

   NOTE 264 — Other SQL-statements executed in <externally-invoked procedure>s in the SQL-client module have the <module authorization identifier> and <schema name> specified or implicit for the SQL-client module.

5) If <schema character set specification> is not specified, then a <schema character set specification> that specifies an implementation-defined character set that contains at least every character that is in <SQL language character> is implicit.

6) If <schema path specification> is not specified, then a <schema path specification> containing an implementation-defined <schema name list> that contains the <schema name> contained in <schema name clause> is implicit.

7) The explicit or implicit <catalog name> of each <schema name> contained in the <schema name list> of the <schema path specification> shall be equivalent to the <catalog name> of the <schema name> contained in the <schema name clause>.

8) The <schema name list> of the explicit or implicit <schema path specification> is used as the SQL-path of the schema. The SQL-path is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in the <schema definition>.

NOTE 265 — <routine name> is defined in Subclause 5.4, “Names and identifiers”.

Access Rules

1) The privileges necessary to execute the <schema definition> are implementation-defined.

General Rules

1) A <schema definition> creates an SQL-schema *S* in a catalog. *S* includes:
   a) A schema name that is equivalent to the explicit or implicit <schema name>.
   b) A schema authorization identifier that is equivalent to the explicit or implicit <authorization identifier>.
   c) A schema character set name that is equivalent to the explicit or implicit <schema character set specification>.
   d) A schema SQL-path that is equivalent to the explicit or implicit <schema path specification>.
   e) The descriptor created by every <schema element> of the <schema definition>.
2) The owner of *S* is schema authorization identifier.
3) The explicit or implicit <character set specification> is used as the default character set used for all <column definition>s and <domain definition>s that do not specify an explicit character set.

Conformance Rules

1) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <schema path specification>.
2) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <schema character set specification>.
3) Without Feature F171, “Multiple schemas per user”, conforming SQL language shall not contain a <schema name clause> that contains a <schema name>.

11.2 <drop schema statement>

This Subclause is modified by Subclause 9.2, "<drop schema statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.2, "<drop schema statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 9.1, "<drop schema statement>", in ISO/IEC 9075-13.

Function

Destroy a schema.

Format

```
<drop schema statement> ::=
  DROP SCHEMA <schema name> <drop behavior>

<drop behavior> ::=
    CASCADE
  | RESTRICT
```

Syntax Rules

1) Let *S* be the schema identified by <schema name>.

2) *S* shall identify a schema in the catalog identified by the explicit or implicit <catalog name>.

3) If RESTRICT is specified, then *S* shall not contain any persistent base tables, global temporary tables, created local temporary tables, views, domains, assertions, character sets, collations, transliterations, triggers, user-defined types, SQL-invoked routines, sequence generators, or roles, and the <schema name> of *S* shall not be generally contained in the SQL routine body of any routine descriptor.

   NOTE 266 — If CASCADE is specified, then such objects will be dropped by the effective execution of the SQL schema manipulation statements specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the <schema name>.

General Rules

1) Let *T* be the <table name> included in the descriptor of any base table or temporary table included in *S*. The following <drop table statement> is effectively executed:

   ```
   DROP TABLE T CASCADE
   ```

2) Let *V* be the <table name> included in the descriptor of any view included in *S*. The following <drop view statement> is effectively executed:

   ```
   DROP VIEW V CASCADE
   ```

3) Let *D* be the <domain name> included in the descriptor of any domain included in *S*. The following <drop domain statement> is effectively executed:

```
DROP DOMAIN D CASCADE
```

4) Let *A* be the <constraint name> included in the descriptor of any assertion included in *S*. The following <drop assertion statement> is effectively executed:

```
DROP ASSERTION A CASCADE
```

5) Let *CD* be the <collation name> included in the descriptor of any collation included in *S*. The following <drop collation statement> is effectively executed:

```
DROP COLLATION CD CASCADE
```

6) Let *TD* be the <transliteration name> included in the descriptor of any transliteration included in *S*. The following <drop transliteration statement> is effectively executed:

```
DROP TRANSLATION TD
```

7) Let *RD* be the <character set name> included in the descriptor of any character set included in *S*. The following <drop character set statement> is effectively executed:

```
DROP CHARACTER SET RD
```

8) Let *DT* be the <user-defined type name> included in the descriptor of any user-defined type included in *S*. The following <drop data type statement> is effectively executed:

```
DROP TYPE DT CASCADE
```

9) Let *TT* be the <trigger name> included in the descriptor of any trigger included in *S*. The following <drop trigger statement> is effectively executed:

```
DROP TRIGGER TT
```

10) For every SQL-invoked routine *R* whose descriptor is included in *S*, let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed for every *R*:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

11) Let *R* be any SQL-invoked routine whose routine descriptor includes an SQL routine body that contains the <schema name> of *S*. Let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

12) Let *SEQN* be the sequence generator name included in the descriptor of any sequence generator included in *S*. The following <drop sequence generator statement> is effectively executed:

```
DROP SEQUENCE SEQN CASCADE
```

13) *S* is destroyed.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop schema statement>.

11.3 <table definition>

This Subclause is modified by Subclause 9.3, "<table definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.3, "<table definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 9.2, "<table definition>", in ISO/IEC 9075-13.

Function

Define a persistent base table, a created local temporary table, or a global temporary table.

Format

```
<table definition> ::=
  CREATE [ <table scope> ] TABLE <table name> <table contents source>
      [ ON COMMIT <table commit action> ROWS ]

<table contents source> ::=
    <table element list>
  | <typed table clause>
  | <as subquery clause>

<table scope> ::=
  <global or local> TEMPORARY

<global or local> ::=
    GLOBAL
  | LOCAL

<table commit action> ::=
    PRESERVE
  | DELETE

<table element list> ::=
  <left paren> <table element> [ { <comma> <table element> }... ] <right paren>

<table element> ::=
    <column definition>
  | <table constraint definition>
  | <like clause>

<typed table clause> ::=
  OF <path-resolved user-defined type name> [ <subtable clause> ]
      [ <typed table element list> ]

<typed table element list> ::=
  <left paren> <typed table element>
      [ { <comma> <typed table element> }... ] <right paren>

<typed table element> ::=
    <column options>
  | <table constraint definition>
  | <self-referencing column specification>

<self-referencing column specification> ::=
  REF IS <self-referencing column name> [ <reference generation> ]
```

```
<reference generation> ::=
    SYSTEM GENERATED
  | USER GENERATED
  | DERIVED

<self-referencing column name> ::=
  <column name>

<column options> ::=
  <column name> WITH OPTIONS <column option list>

<column option list> ::=
  [ <scope clause> ] [ <default clause> ] [ <column constraint definition>... ]

<subtable clause> ::=
  UNDER <supertable clause>

<supertable clause> ::=
  <supertable name>

<supertable name> ::=
  <table name>

<like clause> ::=
  LIKE <table name> [ <like options> ]

<like options> ::=
  <like option>...

<like option> ::=
    <identity option>
  | <column default option>
  | <generation option>

<identity option> ::=
    INCLUDING IDENTITY
  | EXCLUDING IDENTITY

<column default option> ::=
    INCLUDING DEFAULTS
  | EXCLUDING DEFAULTS

<generation option> ::=
    INCLUDING GENERATED
  | EXCLUDING GENERATED

<as subquery clause> ::=
  [ <left paren> <column name list> <right paren> ] AS <subquery>
      <with or without data>

<with or without data> ::=
    WITH NO DATA
  | WITH DATA
```

Syntax Rules

1) The <table contents source> shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, or an <embedded variable specification>.

2) Let *T* be the table defined by the <table definition> *TD*. Let *TN* be the <table name> simply contained in *TD*.

3) If a <table definition> is contained in a <schema definition> *SD* and *TN* contains a <local or schema qualifier>, then that <local or schema qualifier> shall be equivalent to the implicit or explicit <schema name> of *SD*.

4) *TN* shall not identify an existing table descriptor.

5) If the <table definition> is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>. Otherwise, let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *TN*.

6) If <table element> *TEL* is specified, then:

   a) *TEL* shall contain at least one <column definition> or <like clause>.

   b) For each <like clause> *LC* that is directly contained in *TEL*:

      i) Let *LT* be the table identified by the <table name> contained in *LC*.

      ii) If *LT* is a viewed table, then <like options> shall not be specified.

      iii) Let *D* be the degree of *LT*. For *i*, 1 (one) ≤ *i* ≤ *D*:

         1) Let LCD_i, be the column descriptor of the *i*-th column of *LT*.

         2) Let LCN_i be the column name included in LCD_i.

         3) Let LDT_i be the data type included in LCD_i.

         4) If the nullability characteristic included in LCD_i is known not nullable, then let LNC_i be NOT NULL; otherwise, let LNC_i be the zero-length string.

         5) Let CD_i be the <column definition>

            LCN_i LDT_i LNC_i

      iv) If <like options> is specified, then:

         1) <identity option> shall not be specified more than once, <column default option> shall not be specified more than once, and <generation option> shall not be specified more than once.

         2) If <identity option> is not specified, then EXCLUDING IDENTITY is implicit.

         3) If <column default option> is not specified, then EXCLUDING DEFAULTS is implicit.

         4) If <generation option> is not specified, then EXCLUDING GENERATED is implicit.

         5) If INCLUDING IDENTITY is specified and *LT* includes an identity column, then let *ICD* be the column descriptor of that column included in the table descriptor of *LT*. Let *SGD* be the sequence generator descriptor included in *ICD*.

            A) Let *SV* be the start value included in *ICD*.

            B) Let *IV* be the increment included in *SGD*.

C) Let *MAX* be the maximum value included in *SGD*.

D) Let *MIN* be the minimum value included in *SGD*.

E) Let *CYC* be the cycle option included in *SGD*.

F) Let *k* be the ordinal position in which the column described by *ICD* appears in the table identified by *LT*.

G) Case:

I) If *ICD* indicates that values are always generated, then let *G* be GENERATED ALWAYS.

II) If *ICD* indicates that values are generated by default, then let *G* be GENERATED BY DEFAULT.

H) The value of CD_k is replaced by:

```
LCN_k LDT_k
  G AS IDENTITY ( START WITH SV, INCREMENT BY IV,
                  MAXVALUE MAX, MINVALUE MIN, CYC ) LNC_k
```

6) If INCLUDING GENERATED is specified, then let GCD_j, 1 (one) ≤ j ≤ D, be the column descriptors included in the descriptor of *LT*, with *j* being the ordinal position of the column described by GCD_j. For each GCD_j that indicates that the column it describes is a generated column:

A) Let GE_j be the <generation expression> included in GCD_j, where the <table name> contained in any contained <column reference> is replaced by *TN*.

B) The value of CD_j is replaced by

```
LCN_j LDT_j GENERATED ALWAYS AS GE_j LCN_j
```

7) If INCLUDING DEFAULTS is specified, then let DCD_m, 1 (one) ≤ m ≤ D, be the column descriptors included in the descriptor of *LT*, with *m* being the ordinal position of the column described by DCD_m.

For each DCD_m, if DCD_m includes a <default option> DO_m, then the value of CD_m is replaced by

```
LCN_m LDT_m DEFAULT  DO_m LCN_m
```

v) *LC* is effectively replaced by:

```
CD_1, ..., CD_D
```

NOTE 267 — <column constraint>s, except for NOT NULL, are not included in CD_i; <column constraint definition>s are effectively transformed to <table constraint definition>s and are thereby also excluded.

7) If <as subquery clause> is specified, then:

a) Let *ASQ* be the <subquery> immediately contained in <as subquery clause>. Let *QT* be the table specified by *ASQ*.

b) If any two columns in *QT* have equivalent <column name>s, or if any column of *QT* has an implementation-dependent name, then <column name list> shall be specified.

c) Let *D* be the degree of *QT*.

d) <column name list> shall not contain two or more equivalent <column name>s.

e) The number of <column name>s in <column name list> shall be *D*.

f) For *i*, 1 (one) ≤ *i* ≤ *D*:

   i) Case:

      1) If <column name list> is specified, then let QCN_i be the *i*-th <column name> in that <column name list>.

      2) Otherwise, let QCN_i be the <column name> of the *i*-th column of *QT*.

   ii) Let QDT_i be the declared type of the *i*-th column of *QT*.

   iii) If the nullability characteristic of the *i*-th column of *QT* is known not nullable, then let QNC_i be NOT NULL; otherwise, let QNC_i be the zero-length string.

   iv) Let CD_i be the <column definition>

   $QCN_i\ QDT_i\ QNC_i$

g) <as subquery clause> is effectively replaced by a <table element list> *TEL* of the form:

$CD_1, \ldots, CD_D$

8) If <typed table clause> *TTC* is specified, then:

a) The <user-defined type name> simply contained in <path-resolved user-defined type name> shall identify a structured type *ST*.

b) If <subtable clause> is specified, then <self-referencing column specification> shall not be specified. Otherwise, <self-referencing column specification> shall be specified exactly once.

c) If <self-referencing column specification> *SRCS* is specified, then let *RST* be the reference type REF(*ST*).

   i) <subtable clause> shall not be specified.

   ii) <table scope> shall not be specified.

   iii) If SYSTEM GENERATED is specified, then *RST* shall have a system-defined representation.

   iv) If USER GENERATED is specified, then *RST* shall have a user-defined representation.

   v) If DERIVED is specified, then *RST* shall have a derived representation.

   vi) If *RST* has a derived representation, then let *m* be the number of attributes included in the list of attributes of the derived representation of *RST* and let A_i, 1 (one) ≤ *i* ≤ *m*, be those attributes.

1) *TD* shall contain a <table constraint definition> that specifies a <unique constraint definition> *UCD* whose <unique column list> contains the attribute names of A_1, A_2, ..., A_m in that order.

2) If *UCD* does not specify PRIMARY KEY, then for every attribute A_i, 1 (one) ≤ i ≤ m, *TD* shall contain a <column options> CO_i with a <column name> that is equivalent to the <attribute name> of A_i and with a <column constraint definition> that specifies NOT NULL.

vii) Let CD_0 be the <column definition>:

```
CN0 RST SCOPE(TN) UNIQUE NOT NULL
```

where CN_0 denotes the <self-referencing column name> simply contained in *SRCS*.

d) If <subtable clause> is specified, then:

i) The <table name> contained in the <subtable clause> identifies the *direct supertable* of *T*, which shall be a base table. *T* is called a *direct subtable* of the direct supertable of *T*.

ii) *ST* shall be a direct subtype of the structured type of the direct supertable of *T*.

iii) The SQL-schema identified by the explicit or implicit <schema name> of the <table name> of *T* shall include the descriptor of the direct supertable of *T*.

iv) The subtable family of *T* shall not include a member, other than *T* itself, whose associated structured type is *ST*.

v) *TD* shall not contain a <table constraint definition> that specifies PRIMARY KEY.

vi) Let the term *inherited column* of *T* refer to a column of *T* that corresponds to an inherited attribute of *ST*. For every such inherited attribute *IA*, there is a column *CA* of the direct supertable of *T* such that the <column name> of *CA* is equivalent to the <attribute name> of *IA*. *CA* is called the *direct supercolumn* of *IA* in the direct supertable of *T*.

vii) Let CD_0 be the <column definition>:

```
CN0 RST SCOPE(TN) UNIQUE NOT NULL
```

where CN_0 denotes the <self-referencing column name> simply contained in *SRCS*.

e) Let the term *originally-defined column* of *T* refer to a column of *T* that corresponds to an originally-defined attribute of *ST*.

f) Let *n* be the number of attributes of *ST*. Let AD_i, 1 (one) ≤ i ≤ n, be the attribute descriptors included in the data type descriptor of *ST* and let CD_i be the <column definition> CN_i DT_i DC_i, where:

i) CN_i is the attribute name included in AD_i.

ii) DT_i is some <data type> that, under the General Rules of Subclause 6.1, "<data type>", would result in the creation of the data type descriptor included in AD_i.

iii) If AD_i describes an inherited attribute *IA*, then

Case:

1) If the column descriptor of the direct supercolumn of *IA* includes a default value, then DC_i is some <default clause> whose <default option> denotes this default value.

2) Otherwise, DC_i is the zero-length string.

iv) If AD_i describes an originally-defined attribute *OA*, then

Case:

1) If AD_i includes a default value, then DC_i is some <default clause> whose <default option> denotes this default value.

2) Otherwise, DC_i is the zero-length string.

g) If <typed table element list> *TTEL* is specified and <column options> *CO* is specified, then:

i) The <column name> *CN* simply contained in *CO* shall be equivalent to the <column name> CN_j specified in some <column definition> CD_j and shall refer to an originally-defined column of *T*.

ii) *CN* shall not be equivalent to the <column name> simply contained in any other <column options> contained in *TTEL*.

iii) A <column option list> shall immediately contain either a <scope clause> or a <default clause>, or at least one <column constraint definition>.

iv) If *CO* specifies a <scope clause> *SC*, then DT_j shall be a <reference type> *RT*. If *RT* contains a <scope clause>, then that <scope clause> is replaced by *SC*; otherwise, *RT* is replaced by *RT SC*.

NOTE 268 — Changes to the scope of a column of a typed table do not affect the scope defined for the underlying attribute. Such an attribute scope serves as a kind of default for the column's scope, at the time the typed table is defined, and is not restored if a column's scope is dropped.

v) If *CO* specifies a <default clause> *DC*, then DC_j is replaced by *DC* in CD_j.

vi) If *CO* specifies a non-empty list *CCDL* of <column constraint definition>s, then CD_j is replaced by CD_j *CCDL*.

vii) *CO* is deleted from *TTEL*.

h) *T* is a referenceable table.

i) If *TTEL* is empty, then let *TEL* be a <table element list> of the form

CD_0, ..., CD_n

Otherwise, then let *TEL* be a <table element list> of the form

CD_0, ..., CD_n *TTEL*

9) If ON COMMIT is specified, then TEMPORARY shall be specified.

10) If TEMPORARY is specified and ON COMMIT is not specified, then ON COMMIT DELETE ROWS is implicit.

11) Every referenceable table referenced by a <scope clause> contained in a <column definition> or <column options> contained in *TD* shall be

    Case:

    a) If *TD* specifies no <table scope>, then a persistent base table.

    b) If *TD* specifies GLOBAL TEMPORARY, then a global temporary table.

    c) If *TD* specifies LOCAL TEMPORARY, then a created local temporary table.

12) At most one <table element> shall be a <column definition> that contains an <identity column specification>.

13) The scope of the <table name> is the <table definition>, excluding the <as subquery clause>.

Access Rules

1) If a <table definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) If a <like clause> is contained in a <table definition>, then the applicable privileges for *A* shall include SELECT privilege on the table identified in the <like clause>.

3) *A* shall have in its applicable privileges the UNDER privilege on the <supertable name> specified in <subtable clause>.

4) If "OF <path-resolved user-defined type name>" is specified, then the applicable privileges for *A* shall include USAGE on *ST*.

General Rules

1) A <table definition> defines either a persistent base table, a global temporary table or a created local temporary table. If GLOBAL is specified, then a global temporary table is defined. If LOCAL is specified, then a created local temporary table is defined. Otherwise, a persistent base table is defined.

2) The degree of *T* is initially set to 0 (zero); the General Rules of Subclause 11.4, "<column definition>", specify the degree of *T* during the definition of the columns of *T*.

3) If <path-resolved user-defined type name> is specified, then:

    a) Let *R* be the structured type identified by the <user-defined type name> simply contained in <path-resolved user-defined type name>.

    b) *R* is the structured type associated with *T*.

4) A table descriptor *TDS* is created that describes *T*. *TDS* includes:

    a) The table name *TN*.

    b) The column descriptors of every column of *T*, according to the Syntax Rules and General Rules of Subclause 11.4, "<column definition>", applied to the <column definition>s contained in *TEL*, in the order in which they were specified.

    c) If <typed table clause> is specified, then:

        i) An indication that the table is a referenceable table.

ii) An indication that the column at ordinal position 1 (one) is the self-referencing column of *T*. The column descriptor included in *TDS* that describes that column is marked as identifying a self-referencing column.

iii) If *RST* has a system-defined representation, then an indication that the self-referencing column is a system-generated self-referencing column.

iv) If *RST* has a derived representation, then an indication that the self-referencing column is a derived self-referencing column.

v) If *RST* has a user-defined representation, then an indication that the self-referencing column is a user-generated self-referencing column.

d) The table constraint descriptors specified by each <table constraint definition> contained in *TEL*.

e) If a <path-resolved user-defined type name> is specified, then the user-defined type name of *R*.

f) If <subtable clause> is specified, then the table name of the direct supertable of *T* contained in the <subtable clause>.

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

h) A non-empty set of candidate keys.

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

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

k) If TEMPORARY is specified, then

Case:

i) If ON COMMIT PRESERVE ROWS is specified, then the table descriptor includes an indication that ON COMMIT PRESERVE ROWS is specified.

ii) Otherwise, the table descriptor includes an indication that ON COMMIT DELETE ROWS is specified or implied.

l) Case:

i) If <typed table clause> is not specified, then an indication that *T* is insertable-into.

ii) Otherwise,

Case:

1) If the data type descriptor of *R* indicates that *R* is instantiable, then an indication that *T* is insertable-into.

2) Otherwise, an indication that *T* is not insertable-into.

5) In the descriptor of each direct supertable of *T*, *TN* is added to the end of the list of direct subtables.

6) If <subtable clause> is specified, then a set of privilege descriptors is created that defines the privileges SELECT, UPDATE, and REFERENCES for every inherited column of this table to the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <table name>

of the direct supertable from which that column was inherited. These privileges are grantable. The grantor for each of these privilege descriptors is set to the special grantor value "_SYSTEM".

7) A set of privilege descriptors is created that define the privileges INSERT, SELECT, UPDATE, DELETE, TRIGGER, and REFERENCES on this table and SELECT, INSERT, UPDATE, and REFERENCES for every <column definition> in the table definition. If OF <path-resolved user-defined type name> is specified, then a table/method privilege descriptor is created on this table for every method of the structured type identified by the <path-resolved user-defined type name> and the table SELECT privilege has the WITH HIERARCHY OPTION. These privileges are grantable.

   The grantor for each of these privilege descriptors is set to the special grantor value "_SYSTEM". The grantee is <authorization identifier> *A*.

8) If <subtable clause> is specified, then let *ST* be the set of supertables of *T*. Let *PDS* be the set of privilege descriptors that defined SELECT WITH HIERARCHY OPTION privilege on a table in *ST*. For every privilege descriptor in *PDS*, with grantee *G*, grantor *A*,

   Case:

   a) If the privilege is grantable, then let *WGO* be "WITH GRANT OPTION".

   b) Otherwise, let *WGO* be a zero-length string.

   The following <grant statement> is effectively executed without further Access Rule checking:

   ```
   GRANT SELECT ON T TO G WGO FROM A
   ```

9) The row type *RT* of the table *T* defined by the <table definition> is the set of pairs (<field name>, <data type>) where <field name> is the name of a column *C* of *T* and <data type> is the declared type of *C*. This set of pairs contains one pair for each column of *T*, in the order of their ordinal position in *T*.

10) If <as subquery clause> is specified and WITH DATA is specified, then let *QE* be the <query expression> simply contained in *ASQ*. The following <insert statement> is effectively executed without further Access Rule checking:

    ```
    INSERT INTO TN QE
    ```

Conformance Rules

1) Without Feature T171, "LIKE clause in table definition", conforming SQL language shall not contain a <like clause>.

2) Without Feature F531, "Temporary tables", conforming SQL language shall not contain a <table scope> and shall not reference any global or local temporary table.

3) Without Feature S051, "Create table of type", conforming SQL language shall not contain "OF <path-resolved user-defined type name>".

4) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <column option list> that contains a <scope clause>.

5) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain <reference generation> that does not contain SYSTEM GENERATED.

6) Without Feature S081, "Subtables", conforming SQL language shall not contain a <subtable clause>.

7) Without Feature T172, “AS subquery clause in table definition”, conforming SQL language shall not contain an <as subquery clause>.

8) Without Feature T173, “Extended LIKE clause in table definition”, a <like clause> shall not contain <like options>.

11.4 <column definition>

This Subclause is modified by Subclause 12.1, "<column definition>", in ISO/IEC 9075-14.

Function

Define a column of a base table.

Format

```
<column definition> ::=
  <column name> [ <data type or domain name> ]
      [ <default clause> | <identity column specification> | <generation clause> ]
      [ <column constraint definition>... ]
      [ <collate clause> ]

<data type or domain name> ::=
    <data type>
  | <domain name>

<column constraint definition> ::=
  [ <constraint name definition> ] <column constraint> [ <constraint characteristics> ]

<column constraint> ::=
    NOT NULL
  | <unique specification>
  | <references specification>
  | <check constraint definition>

<identity column specification> ::=
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
      [ <left paren> <common sequence generator options> <right paren> ]

<generation clause> ::=
  <generation rule> AS <generation expression>

<generation rule> ::=
  GENERATED ALWAYS

<generation expression> ::=
  <left paren> <value expression> <right paren>
```

Syntax Rules

1) The <column definition> shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, or an <embedded variable specification>.

2) Case:

   a) If the <column definition> is contained in a <table definition>, then let *T* be the table defined by that <table definition>.

b) If the <column definition> is contained in a <temporary table declaration>, then let *T* be the table declared by that <temporary table declaration>.

c) If the <column definition> is contained in an <alter table statement>, then let *T* be the table identified in the containing <alter table statement>.

The <column name> in the <column definition> shall not be equivalent to the <column name> of any other column of *T*.

3) Let *A* be the <authorization identifier> that owns *T*.

4) Let *C* be the <column name> of the <column definition>.

5) <data type or domain name> shall unambiguously reference either a <data type> or a <domain name>.

6) If <domain name> is specified, then let *D* be the domain identified by the <domain name>.

7) If <generation clause> *GC* is specified, then:

a) Let *GE* be the <generation expression> contained in *GC*.

b) *C* is a generated column.

c) Every <column reference> contained in *GE* shall reference a base column of *T*.

d) *GE* shall be deterministic.

e) *GE* shall not contain a <routine invocation> whose subject routine possibly reads SQL-data.

f) *GE* shall not contain a <query expression>.

8) If <generation clause> is omitted, then either <data type> or <domain name> shall be specified.

9) Case:

a) If <column definition> immediately contains <domain name>, then it shall not also immediately contain <collate clause>.

b) Otherwise, <collate clause> shall not be both specified in <data type> and immediately contained in <column definition>. If <collate clause> is immediately contained in <column definition>, then it is equivalent to specifying an equivalent <collate clause> in <data type>.

10) The declared type of the column is

Case:

a) If <data type> is specified, then that data type. If <generation clause> is also specified, then the declared type of <generation expression> shall be assignable to the declared type of the column.

b) If <domain name> is specified, then the declared type of *D*. If <generation clause> is also specified, then the declared type of <generation expression> shall be assignable to the declared type of the column.

c) If <generation clause> is specified, then the declared type of *GE*.

11) If a <data type> is specified, then:

a) Let *DT* be the <data type>.

b) If *DT* specifies CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT and does not specify a <character set specification>, then the <character set specification> specified or

implicit in the <schema character set specification> of the <schema definition> that created the schema identified by the <schema name> immediately contained in the <table name> of the containing <table definition> or <alter table statement> is implicit.

12) If <identity column specification> *ICS* is specified, then:

a) Case:

i) If the declared type of the column being defined is a distinct type *DIST*, then the source type of *DIST* shall be exact numeric with scale 0 (zero). Let *ICT* be the source type of *DIST*.

ii) Otherwise, the declared type of the column being defined shall be exact numeric with scale 0 (zero). Let *ICT* be the declared type of the column being defined.

b) Let *SGO* be the <common sequence generator options>.

c) The Syntax Rules of Subclause 9.24, "Creation of a sequence generator", are applied with *SGO* as *OPTIONS* and *ICT* as *DATA TYPE*.

d) The <column constraint definition> NOT NULL NOT DEFERRABLE is implicit.

13) If a <column constraint definition> is specified, then let *CND* be the <constraint name definition> if one is specified and let *CND* be a zero-length string otherwise; let *CA* be the <constraint characteristics> if specified and let *CA* be a zero-length string otherwise. The <column constraint definition> is equivalent to a <table constraint definition> as follows.

Case:

a) If a <column constraint definition> is specified that contains the <column constraint> NOT NULL, then it is equivalent to the following <table constraint definition>:

```
CND CHECK ( C IS NOT NULL ) CA
```

b) If a <column constraint definition> is specified that contains a <unique specification> *US*, then it is equivalent to the following <table constraint definition>:

```
CND US (C) CA
```

NOTE 269 — The <unique specification> is defined in Subclause 11.7, "<unique constraint definition>".

c) If a <column constraint definition> is specified that contains a <references specification> *RS*, then it is equivalent to the following <table constraint definition>:

```
CND FOREIGN KEY (C) RS CA
```

NOTE 270 — The <references specification> is defined in Subclause 11.8, "<referential constraint definition>".

d) If a <column constraint definition> is specified that contains a <check constraint definition> *CCD*, then it is equivalent to the following <table constraint definition>:

```
CND CCD CA
```

Each column reference directly contained in the <search condition> shall reference column *C*.

14) The schema identified by the explicit or implicit qualifier of the <domain name> shall include the descriptor of *D*.

Access Rules

1) If <domain name> is specified, then the applicable privileges for *A* shall include USAGE on *D*.

General Rules

1) A <column definition> defines a column in a table.

2) If the <column definition> specifies <data type>, then a data type descriptor is created that describes the declared type of the column being defined.

3) The degree of the table *T* being defined in the containing <table definition> or <temporary table declaration>, or being altered by the containing <alter table statement> is increased by 1 (one).

4) A column descriptor is created that describes the column being defined. The column descriptor includes:

   a) *C*, the name of the column.

   b) Case:

      i) If the <column definition> specifies a <data type> or a <generation clause>, then the data type descriptor of the declared type of the column.

      ii) Otherwise, the <domain name>that identifies the domain of the column.

   c) The ordinal position of the column, which is equal to the degree of *T*.

   d) The nullability characteristic of the column, determined according to the rules in Subclause 4.13, "Columns, fields, and attributes".

      NOTE 271 — Both <column constraint definition>s and <table constraint definition>s shall be analyzed to determine the nullability characteristics of all columns.

   e) If <default clause> is specified, then the <default option>.

   f) If <identity column specification> is specified, then:

      i) An indication that the column is an identity column.

      ii) If ALWAYS is specified, then an indication that values are always generated.

      iii) If BY DEFAULT is specified, then an indication that values are generated by default.

      iv) The descriptor of the sequence generator descriptor *SG* resulting from application of the General Rules of Subclause 9.24, "Creation of a sequence generator", with *SGO* as *OPTIONS* and *ICT* as *DATA TYPE*.

      v) The next available value of *SG* as the start value.

   g) Case:

      i) If the <column definition> specifies a <generation clause>, then an indication that the column is "ALWAYS" generated and *GE*.

      ii) Otherwise, an indication that the column is "NEVER" generated.

   h) An indication that the column is updatable.

5) If <domain name> is specified, then, for every domain constraint descriptor *DCD* included in the domain descriptor of *D*, let *DSC* be the template <search condition> included in *DCD*. Let *CSC* be a copy of *DSC* in which every instance of the <general value specification> VALUE is replaced by *C*. A domain constraint usage descriptor is created and added to the set of domain constraint usage descriptors included in *DCD*. The domain constraint usage descriptor created includes:

   a) The name of the applicable column.

   b) The constraint name included in *DCD*, as the name of the applicable domain constraint.

   c) The indication included in *DCD* as to whether the constraint is deferrable.

   d) The initial constraint mode included in *DCD*.

   e) The applicable <search condition>

```
( SELECT EVERY ( CSC )
  FROM T )
```

   NOTE 272 — This is a <scalar subquery> of declared type BOOLEAN.

Conformance Rules

1) Without Feature F692, “Extended collation support”, conforming SQL language shall not contain a <column definition> that immediately contains a <collate clause>.

2) Without Feature T174, “Identity columns”, conforming SQL language shall not contain an <identity column specification>.

3) Without Feature T175, “Generated columns”, conforming SQL language shall not contain a <generation clause>.

11.5 <default clause>

This Subclause is modified by Subclause 9.5, "<default clause>", in ISO/IEC 9075-4.

Function

Specify the default for a column, domain, or attribute.

Format

```
<default clause> ::=
  DEFAULT <default option>

<default option> ::=
    <literal>
  | <datetime value function>
  | USER
  | CURRENT_USER
  | CURRENT_ROLE
  | SESSION_USER
  | SYSTEM_USER
  | CURRENT_CATALOG
  | CURRENT_SCHEMA
  | CURRENT_PATH
  | <implicitly typed value specification>
```

Syntax Rules

1) The subject data type of a <default clause> is the data type specified in the descriptor identified by the containing <column definition>, <domain definition>, <attribute definition>, <alter column definition>, or <alter domain statement>.

2) If USER is specified, then CURRENT_USER is implicit.

3) Case:

   a) If the subject data type of the <default clause> is a user-defined type, a reference type, or a row type, then <default option> shall specify <null specification>.

   b) If the subject data type of the <default clause> is a collection type, then <default option> shall specify <implicitly typed value specification>. If the <default option> specifies an <empty specification> that specifies ARRAY, then the subject data type shall be an array type. If the <default option> specifies an <empty specification> that specifies MULTISET, then the subject data type shall be a multiset type.

4) Case:

   a) If a <literal> is specified, then

      Case:

      i) If the subject data type is character string, then the <literal> shall be a <character string literal>. If the length of the subject data type is fixed, then the length in characters of the <character string literal> shall not be greater than the length of the subject data type. If the length of the

subject data type is variable, then the length in characters of the <character string literal> shall not be greater than the maximum length of the subject data type. The <literal> shall have the same character repertoire as the subject data type.

ii) If the subject data type is binary string, then the <literal> shall be a <binary string literal>. If the length of the subject data type is fixed, then the length in octets of the <character string literal> shall not be greater than the length of the subject data type. If the length of the subject data type is variable, then the length in octets of the <binary string literal> shall not be greater than the maximum length of the subject data type.

iii) If the subject data type is exact numeric, then the <literal> shall be a <signed numeric literal> that simply contains an <exact numeric literal>. There shall be a representation of the value of the <literal> in the subject data type that does not lose any significant digits.

iv) If the subject data type is approximate numeric, then the <literal> shall be a <signed numeric literal>.

v) If the subject data type is datetime, then the <literal> shall be a <datetime literal> with the same primary datetime fields and the same time zone datetime fields as the subject data type. If SECOND is one of these fields, then the fractional seconds precision of the <datetime literal> shall be less than or equal to the fractional seconds precision of the subject data type.

vi) If the subject data type is interval, then the <literal> shall be an <interval literal> and shall contain the same <interval qualifier> as the subject data type.

vii) If the subject data type is boolean, then the <literal> shall be a <boolean literal>.

b) If CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, or CURRENT_SCHEMA is specified, then the subject data type shall be character string with character set SQL_IDENTIFIER. If the length of the subject data type is fixed, then its length shall not be less than 128 characters. If the length of the subject data type is variable, then its maximum length shall not be less than 128 characters.

c) If CURRENT_PATH is specified, then the subject data type shall be character string with character set SQL_IDENTIFIER. If the length of the subject data type is fixed, then its length shall not be less than 1031 characters. If the length of the subject data type is variable, then its maximum length shall not be less than 1031 characters.

d) If <datetime value function> is specified, then the subject data type shall be datetime with the same declared datetime data type of the <datetime value function>.

e) If <empty specification> is specified, then the subject data type shall be a collection type.

Access Rules

None.

General Rules

1) The default value inserted in the column descriptor, if the <default clause> is to apply to a column, or in the domain descriptor, if the <default clause> is to apply to a domain, or in the attribute descriptor, if the <default clause> is to apply to an attribute, is the <default option>.

2) The value specified by a <default option> is

   Case:

   a) If the <default option> contains a <literal>, then

      Case:

      i) If the subject data type is numeric, then the numeric value of the <literal>.

      ii) If the subject data type is character string with variable length, then the value of the <literal>.

      iii) If the subject data type is character string with fixed length, then the value of the <literal>, extended as necessary on the right with <space>s to the length in characters of the subject data type.

      iv) If the subject data type is fixed-length binary string, then the value of the <literal>, extended as necessary to the right with X'00's to the length in octets of the subject data type.

      v) If the subject data type is variable-length binary string, then the value of the <literal>.

      vi) If the subject data type is datetime or interval, then the value of the <literal>.

      vii) If the subject data type is boolean, then the value of the <literal>.

   b) If the <default option> specifies CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH, then

      Case:

      i) If the subject data type is character string with variable length, then the value obtained by an evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH at the time that the default value is required.

      ii) If the subject data type is character string with fixed length, then the value obtained by an evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH at the time that the default value is required, extended as necessary on the right with <space>s to the length in characters of the subject data type.

   c) If the <default option> contains a <datetime value function>, then the value of an evaluation of the <datetime value function> at the time that the default value is required.

   d) If the <default option> specifies <empty specification>, then an empty collection.

3) When a site *S* is set to its default value,

   Case:

   a) If the descriptor of *S* indicates that it represents a column of which some underlying column is an identity column or a generated column, then *S* is marked as *unassigned*.

      NOTE 273 — The notion of a site being unassigned is only for definitional purposes in this International Standard. It is not a state that can persist so as to be visible in SQL-data. The treatment of unassigned sites is given in Subclause 15.10, "Effect of inserting tables into base tables", and Subclause 15.13, "Effect of replacing rows in base tables".

   b) If the data descriptor for the site includes a <default option>, then *S* is set to the value specified by that <default option>.

c) If the data descriptor for the site includes a <domain name> that identifies a domain descriptor that includes a <default option>, then *S* is set to the value specified by that <default option>.

d) If the default value is for a column *C* of a candidate row for insertion into or update of a derived table *DT* and *C* has a single counterpart column *CC* in a leaf generally underlying table of *DT*, then *S* is set to the default value of *CC*, which is obtained by applying the General Rules of this Subclause.

e) Otherwise, *S* is set to the null value.

NOTE 274 — If <default option> specifies CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH, then the "value in the column descriptor" will effectively be the text of the <default option>, whose evaluation occurs at the time that the default value is required.

4) If the <default clause> is contained in an <SQL schema statement> and character representation of the <default option> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — default value too long for information schema*.

NOTE 275 — The Information Schema is defined in ISO/IEC 9075-11.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <default option> that contains CURRENT_PATH.

2) Without Feature F321, "User authorization", conforming SQL language shall not contain a <default option> that contains CURRENT_USER, SESSION_USER, or SYSTEM_USER.

NOTE 276 — Although CURRENT_USER and USER are semantically the same, without Feature F321, "User authorization", CURRENT_USER shall be specified as USER.

3) Without Feature T332, "Extended roles", conforming SQL language shall not contain a <default option> that contains CURRENT_ROLE.

4) Without Feature F762, "CURRENT_CATALOG", conforming SQL language shall not contain a <default option> that contains CURRENT_CATALOG.

5) Without Feature F763, "CURRENT_SCHEMA", conforming SQL language shall not contain a <default option> that contains CURRENT_SCHEMA.

11.6 <table constraint definition>

Function

Specify an integrity constraint.

Format

```
<table constraint definition> ::=
  [ <constraint name definition> ] <table constraint>
      [ <constraint characteristics> ]

<table constraint> ::=
    <unique constraint definition>
  | <referential constraint definition>
  | <check constraint definition>
```

Syntax Rules

1) If <constraint characteristics> is not specified, then INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

2) If <constraint name definition> is specified and its <constraint name> contains a <schema name>, then that <schema name> shall be equivalent to the explicit or implicit <schema name> of the <table name> of the table identified by the containing <table definition> or <alter table statement>.

3) If <constraint name definition> is not specified, then a <constraint name definition> that contains an implementation-dependent <constraint name> is implicit. The assigned <constraint name> shall obey the Syntax Rules of an explicit <constraint name>.

4) Let *S* be the schema identified by the explicit or implicit <schema name> of the <constraint name>. *S* shall not include a constraint descriptor whose constraint name is <constraint name>.

Access Rules

None.

General Rules

1) A <table constraint definition> defines a table constraint.

2) A table constraint descriptor is created that describes the table constraint being defined. The table constraint descriptor includes:

   a) The <constraint name> contained in the explicit or implicit <constraint name definition>.

   b) An indication of whether the constraint is deferrable or not deferrable.

   c) An indication of whether the initial constraint mode of the constraint is deferred or immediate.

d) The applicable <search condition> as specified in the General Rules applicable to the particular kind of <table constraint> contained in the <table constraint definition>, and any additional contents specified by those General Rules.

3) If the <table constraint> is a <check constraint definition>, then let *SC* be the <search condition> immediately contained in the <check constraint definition> and let *T* be the table name included in the corresponding table constraint descriptor; the applicable <search condition> included in the table constraint descriptor is:

```
( SELECT EVERY ( SC )
  FROM T )
```

NOTE 277 — This is a <scalar subquery> of declared type BOOLEAN.

Conformance Rules

None.

11.7 <unique constraint definition>

This Subclause is modified by Subclause 11.4, "<unique constraint definition>", in ISO/IEC 9075-9.

Function

Specify a uniqueness constraint for a table.

Format

```
<unique constraint definition> ::=
    <unique specification> <left paren> <unique column list> <right paren>
  | UNIQUE ( VALUE )

<unique specification> ::=
    UNIQUE
  | PRIMARY KEY

<unique column list> ::=
  <column name list>
```

Syntax Rules

1) Each column identified by a <column name> in the <unique column list> is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, "Grouping operations", apply.

2) Let *T* be the table identified by the containing <table definition> or <alter table statement>. Let *TN* be the <table name> of *T*.

3) If <unique column list> *UCL* is specified, then

   a) Each <column name> in the <unique column list> shall identify a column of *T*, and the same column shall not be identified more than once.

   b) The set of columns in the <unique column list> shall be distinct from the unique columns of any other unique constraint descriptor that is included in the base table descriptor of *T*.

   c) Case:

      i) If the <unique specification> specifies PRIMARY KEY, then let *SC* be the <search condition>:

```
UNIQUE ( SELECT UCL
         FROM TN )
  AND
( SELECT EVERY ( UCL ) IS NOT NULL )
  FROM TN )
```

NOTE 278 — The second operand of AND in this expression is a <scalar subquery> of declared type BOOLEAN.

      ii) Otherwise, let *SC* be the <search condition>:

```
UNIQUE ( SELECT UCL FROM TN )
```

4) If UNIQUE (VALUE) is specified, then let *SC* be the <search condition>:

```
UNIQUE ( SELECT TN.* FROM TN )
```

5) If the <unique specification> specifies PRIMARY KEY, then for each <column name> in the explicit or implicit <unique column list> for which NOT NULL is not specified, NOT NULL is implicit in the <column definition>.

6) A <table definition> shall specify at most one implicit or explicit <unique constraint definition> that specifies PRIMARY KEY.

7) If a <unique constraint definition> that specifies PRIMARY KEY is contained in an <add table constraint definition>, then the table identified by the <table name> immediately contained in the containing <alter table statement> shall not have a unique constraint that was defined by a <unique constraint definition> that specified PRIMARY KEY.

Access Rules

None.

General Rules

1) A <unique constraint definition> defines a unique constraint.

2) The applicable <search condition> included in the table constraint descriptor created as a result of execution of the containing <table constraint definition> is *SC*. This descriptor additionally includes:

   a) The <unique specification>, indicating whether the constraint is defined with PRIMARY KEY or UNIQUE.

   b) The names of the unique columns specified in the <unique column list>.

3) The unique constraint is not satisfied if and only if

```
EXISTS ( SELECT * FROM TN WHERE NOT ( SC ) )
```

   is *True*.

Conformance Rules

1) Without Feature S291, “Unique constraint on entire row”, conforming SQL language shall not contain UNIQUE(VALUE).

2) Without Feature T591, “UNIQUE constraints of possibly null columns”, in conforming SQL language, if UNIQUE is specified, then the <column definition> for each column whose <column name> is contained in the <unique column list> shall contain NOT NULL.

NOTE 279 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

11.8 <referential constraint definition>

Function

Specify a referential constraint.

Format

```
<referential constraint definition> ::=
  FOREIGN KEY <left paren> <referencing columns> <right paren>
      <references specification>

<references specification> ::=
  REFERENCES <referenced table and columns>
      [ MATCH <match type> ] [ <referential triggered action> ]

<match type> ::=
    FULL
  | PARTIAL
  | SIMPLE

<referencing columns> ::=
  <reference column list>

<referenced table and columns> ::=
  <table name> [ <left paren> <reference column list> <right paren> ]

<reference column list> ::=
  <column name list>

<referential triggered action> ::=
    <update rule> [ <delete rule> ]
  | <delete rule> [ <update rule> ]

<update rule> ::=
  ON UPDATE <referential action>

<delete rule> ::=
  ON DELETE <referential action>

<referential action> ::=
    CASCADE
  | SET NULL
  | SET DEFAULT
  | RESTRICT
  | NO ACTION
```

Syntax Rules

1) If <match type> is not specified, then SIMPLE is implicit.

2) Let *referencing table* be the table identified by the containing <table definition> or <alter table statement>. Let *referenced table* be the table identified by the <table name> in the <referenced table and columns>.

Let *referencing columns* be the column or columns identified by the <reference column list> in the <referencing columns> and let *referencing column* be one such column.

3) Case:

   a) If the <referenced table and columns> specifies a <reference column list>, then there shall be a one-to-one correspondence between the set of <column name>s contained in that <reference column list> and the set of <column name>s contained in the <unique column list> of a unique constraint of the referenced table such that corresponding <column name>s are equivalent. Let *referenced columns* be the column or columns identified by that <reference column list> and let *referenced column* be one such column. Each referenced column shall identify a column of the referenced table and the same column shall not be identified more than once.

   b) If the <referenced table and columns> does not specify a <reference column list>, then the table descriptor of the referenced table shall include a unique constraint that specifies PRIMARY KEY. Let *referenced columns* be the column or columns identified by the unique columns in that unique constraint and let *referenced column* be one such column. The <referenced table and columns> shall be considered to implicitly specify a <reference column list> that is identical to that <unique column list>.

4) The table constraint descriptor describing the <unique constraint definition> whose <unique column list> identifies the referenced columns shall indicate that the unique constraint is not deferrable.

5) The referenced table shall be a base table.

   Case:

   a) If the referencing table is a persistent base table, then the referenced table shall be a persistent base table.

   b) If the referencing table is a global temporary table, then the referenced table shall be a global temporary table.

   c) If the referencing table is a created local temporary table, then the referenced table shall be either a global temporary table or a created local temporary table.

   d) If the referencing table is a declared local temporary table, then the referenced table shall be either a global temporary table, a created local temporary table or a declared local temporary table.

6) If the referenced table is a temporary table with ON COMMIT DELETE ROWS specified, then the referencing table shall specify ON COMMIT DELETE ROWS.

7) Each referencing column shall identify a column of the referencing table, and the same column shall not be identified more than once.

8) Each referencing column is an operand of a grouping operation. The Syntax Rules of Subclause 9.10, “Grouping operations”, apply.

9) The <referencing columns> shall contain the same number of <column name>s as the <referenced table and columns>. The *i*-th column identified in the <referencing columns> corresponds to the *i*-th column identified in the <referenced table and columns>. The declared type of each referencing column shall be comparable to the declared type of the corresponding referenced column. There shall not be corresponding constituents of the declared type of a referencing column and the declared type of the corresponding referenced column such that one constituent is datetime with time zone and the other is datetime without time zone.

10) If a <referential constraint definition> does not specify any <update rule>, then an <update rule> with a <referential action> of NO ACTION is implicit.

11) If a <referential constraint definition> does not specify any <delete rule>, then a <delete rule> with a <referential action> of NO ACTION is implicit.

12) If any referencing column is a generated column, then:

    a) <referential action> shall not specify SET NULL or SET DEFAULT.

    b) <update rule> shall not specify ON UPDATE CASCADE.

13) Let *T* be the referenced table. The schema identified by the explicit or implicit qualifier of the <table name> shall include the descriptor of *T*.

Access Rules

1) The applicable privileges for the owner of *T* shall include REFERENCES for each referenced column.

General Rules

1) A <referential constraint definition> defines a referential constraint.

2) Let R_f be the referencing columns in the referencing table *U* and let R_t be the referenced columns in the referenced table *T*. Let *MP* be

   Case:

   a) If SIMPLE is specified or implicit, then

      ```
      Rf MATCH SIMPLE ( SELECT Rt FROM T )
      ```

   b) If PARTIAL is specified, then

      ```
      Rf MATCH PARTIAL ( SELECT Rt FROM T )
      ```

   c) If FULL is specified, then

      ```
      Rf MATCH FULL ( SELECT Rt FROM T )
      ```

3) The applicable <search condition> included in the table constraint descriptor created as a result of executing the containing <table constraint definition> is:

   ```
   ( SELECT EVERY ( MP )
     FROM U )
   ```

   NOTE 280 — This is a <scalar subquery> of declared type BOOLEAN.

   This descriptor additionally includes:

   a) A list of the names of the referencing columns specified in the <referencing columns>, in an implementation-defined order.

   b) The name of the referenced table specified in the <referenced table and columns>.

c) A list of the names of the referenced columns specified in the <referenced table and columns>, ordered such that each element is in the same position as that of its corresponding referencing column name in the list of names of referencing columns.

d) The value of the <match type>, if specified.

e) The <referential triggered action>, if specified.

4) Case:

a) If SIMPLE is specified or implicit, or if FULL is specified, then for a given row in the referenced table, every row that is a subrow or a superrow of a row *R* in the referencing table such that the referencing column values equal the corresponding referenced column values in *R* for the referential constraint is a *matching row*.

b) If PARTIAL is specified, then:

i) For a given row in the referenced table, every row that is a subrow or a superrow of a row *R* in the referencing table such that *R* has at least one non-null referencing column value and the non-null referencing column values of *R* equal the corresponding referenced column values for the referential constraint is a *matching row*.

ii) For a given row in the referenced table, every matching row for that given row that is a matching row only to the given row in the referenced table for the referential constraint is a *unique matching row*. For a given row in the referenced table, a matching row for that given row that is not a unique matching row for that given row for the referential constraint is a *non-unique matching row*.

5) For each row of the referenced table, its matching rows, unique matching rows, and non-unique matching rows are determined immediately prior to the execution of any <SQL procedure statement>. No new matching rows are added during the execution of that <SQL procedure statement>.

The association between a referenced row and a non-unique matching row is dropped during the execution of that SQL-statement if the referenced row is either marked for deletion or updated to a distinct value on any referenced column that corresponds to a non-null referencing column. This occurs immediately after such a mark for deletion or update of the referenced row. Unique matching rows and non-unique matching rows for a referenced row are evaluated immediately after dropping the association between that referenced row and a non-unique matching row.

6) Let *CTEC* be the current trigger execution context. Let *SSC* be the set of state changes in *CTEC*. Let SC_i be a state change in *SSC*.

7) Let *F* be the referencing table.

8) For every row of the referenced table that is marked for deletion and has not previously been marked for deletion,

Case:

a) If SIMPLE is specified or implicit, or if FULL is specified, then

Case:

i) If the <delete rule> specifies CASCADE, then:

1) *F* is identified for deletion processing and every matching row in *F* is identified for deletion.

2) The General Rules of Subclause 15.7, "Effect of deleting rows from base tables" are applied.

ii) If the <delete rule> specifies SET NULL, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the null value. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iii) If the <delete rule> specifies SET DEFAULT, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iv) If the <delete rule> specifies RESTRICT and there exists some matching row, then an exception condition is raised: *integrity constraint violation — restrict violation*.

b) If PARTIAL is specified, then

Case:

i) If the <delete rule> specifies CASCADE, then:

1) *F* is identified for deletion processing and every unique matching row in *F* is identified for deletion.

2) The General Rules of Subclause 15.7, "Effect of deleting rows from base tables" are applied.

ii) If the <delete rule> specifies SET NULL, then:

1) Each unique matching row *UMR* in *F* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting each referencing column in the copy to the null value. *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iii) If the <delete rule> specifies SET DEFAULT, then:

1) Each unique matching row *UMR* in *F* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting each referencing column in the copy to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *UMR* is identified

for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iv) If the <delete rule> specifies RESTRICT and there exists some unique matching row, then an exception condition is raised: *integrity constraint violation — restrict violation*.

NOTE 281 — Otherwise, the <referential action> is not performed.

9) If a non-null value of a referenced column *RC* in the referenced table is updated to a value that is distinct from the current value of *RC*, then, for every member *F* of the subtable family of the referencing table

Case:

a) If SIMPLE is specified or implicit, or if FULL is specified, then

Case:

i) If the <update rule> specifies CASCADE, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the new value of that referenced column. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

ii) If the <update rule> specifies SET NULL, then

Case:

1) If SIMPLE is specified or implicit, then:

A) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the null value. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

B) *F* is identified for replacement processing with subtables with respect to the referencing columns.

C) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

2) If <match type> specifies FULL, then:

A) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the null value. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

B) *F* is identified for replacement processing with subtables with respect to the referencing columns.

C) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iii) If the <update rule> specifies SET DEFAULT, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iv) If the <update rule> specifies RESTRICT and there exists some matching row, then an exception condition is raised: *integrity constraint violation — restrict violation*.

b) If PARTIAL is specified, then

Case:

i) If the <update rule> specifies CASCADE, then:

1) Each unique matching row *UMR* in *F* that contains a non-null value in the referencing column *C1* in *F* that corresponds to the updated referenced column *C2* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting *C1* in the copy to the new value *V* of *C2*, provided that, in all updated rows in the referenced table that formerly had, during the same execution of the same innermost SQL-statement, that unique matching row as a matching row, the values in *C2* have all been updated to a value that is not distinct from *V*. If this last condition is not satisfied, then an exception condition is raised: *triggered data change violation*. *UMR* is identified for replacement by *NUMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

NOTE 282 — Because of the Rules of Subclause 8.2, "<comparison predicate>", on which the definition of "distinct" relies, the values in *C2* may have been updated to values that are not distinct, yet are not identical. Which of these non-distinct values is used for the cascade operation is implementation-dependent.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

ii) If the <update rule> specifies SET NULL, then:

1) Each unique matching row *UMR* in *F* that contains a non-null value in the referencing column in *F* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting that referencing column in the copy to the null value. *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iii) If the <update rule> specifies SET DEFAULT, then:

1) Each unique matching row *UMR* in *F* that contains a non-null value in the referencing column in *F* that corresponds with the updated referenced column is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting that referencing column in the copy to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

iv) If the <update rule> specifies RESTRICT and there exists some unique matching row, then an exception condition is raised: *integrity constraint violation — restrict violation*.

NOTE 283 — Otherwise, the <referential action> is not performed.

10) Let *ISS* be the innermost SQL-statement being executed.

11) If evaluation of these General Rules during the execution of *ISS* would cause an update of some site to a value that is distinct from the value to which that site was previously updated during the execution of *ISS*, then an exception condition is raised: *triggered data change violation*.

12) If evaluation of these General Rules during the execution of *ISS* would cause deletion of a row containing a site that is identified for replacement in that row, then an exception condition is raised: *triggered data change violation*.

13) If evaluation of these General Rules during the execution of *ISS* would cause either an attempt to update a row that has been deleted by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or that has been updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, or an attempt to mark for deletion such a row, then a completion condition is raised: *warning — cursor operation conflict*.

14) For every row *RMD* that is marked for deletion, every subrow of *RMD* and every superrow of *RMD* is marked for deletion.

15) If any table *T* is the subject table of a state change in *SSC* that has been created or modified during evaluation of the preceding General Rules of this subclause, then, for every referential constraint descriptor, the preceding General Rules of this subclause are applied.

NOTE 284 — Thus these rules are repeatedly evaluated until no further transitions are generated.

Conformance Rules

1) Without Feature T191, "Referential action RESTRICT", conforming SQL language shall not contain a <referential action> that contains RESTRICT.

2) Without Feature F741, "Referential MATCH types", conforming SQL language shall not contain a <references specification> that contains MATCH.

3) Without Feature F191, "Referential delete actions", conforming SQL language shall not contain a <delete rule>.

4) Without Feature F701, “Referential update actions”, conforming SQL language shall not contain an <update rule>.

5) Without Feature T201, “Comparable data types for referential constraints”, conforming SQL language shall not contain a <referencing columns> in which the data type of each referencing column is not the same as the data type of the corresponding referenced column.

NOTE 285 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

11.9 <check constraint definition>

This Subclause is modified by Subclause 9.6, "<check constraint definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.5, "<check constraint definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 12.2, "<check constraint definition>", in ISO/IEC 9075-14.

Function

Specify a condition for the SQL-data.

Format

```
<check constraint definition> ::=
  CHECK <left paren> <search condition> <right paren>
```

Syntax Rules

1) The <search condition> shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, or an <embedded variable specification>.

2) The <search condition> shall not contain a <set function specification> that is not contained in a <query expression>.

3) If <check constraint definition> is contained in a <table definition> or <alter table statement>, then let *T* be the table identified by the containing <table definition> or <alter table statement>.

   Case:

   a) If *T* is a persistent base table, or if the <check constraint definition> is contained in a <domain definition> or <alter domain statement>, then no <table reference> generally contained in the <search condition> shall reference a temporary table.

   b) If *T* is a global temporary table, then no <table reference> generally contained in the <search condition> shall reference a table other than a global temporary table.

   c) If *T* is a created local temporary table, then no <table reference> generally contained in the <search condition> shall reference a table other than either a global temporary table or a created local temporary table.

   d) If *T* is a declared local temporary table, then no <table reference> generally contained in the <search condition> shall reference a persistent base table.

4) If the <check constraint definition> is contained in a <table definition> that defines a temporary table and specifies ON COMMIT PRESERVE ROWS or a <temporary table declaration> that specifies ON COMMIT PRESERVE ROWS, then the <search condition> shall not contain a reference to a temporary table defined by a <table definition> or a <temporary table declaration> that specifies ON COMMIT DELETE ROWS.

5) The <search condition> shall simply contain a <boolean value expression> that is retrospectively deterministic.

   NOTE 286 — "retrospectively deterministic" is defined in Subclause 6.34, "<boolean value expression>".

6) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

7) Let *A* be the <authorization identifier> that owns *T*.

Access Rules

None.

General Rules

1) A <check constraint definition> defines a check constraint.

2) If the character representation of the <search condition> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — search condition too long for information schema*.

NOTE 287 — The Information Schema is defined in ISO/IEC 9075-11.

Conformance Rules

1) Without Feature F671, “Subqueries in CHECK constraints”, conforming SQL language shall not contain a <search condition> contained in a <check constraint definition> that contains a <query expression>.

2) Without Feature F672, “Retrospective check constraints”, conforming SQL language shall not contain a <check constraint definition> that generally contains CURRENT_DATE, CURRENT_TIMESTAMP, or LOCALTIMESTAMP.

11.10 <alter table statement>

Function

Change the definition of a table.

Format

```
<alter table statement> ::=
  ALTER TABLE <table name> <alter table action>

<alter table action> ::=
    <add column definition>
  | <alter column definition>
  | <drop column definition>
  | <add table constraint definition>
  | <drop table constraint definition>
```

Syntax Rules

1) Let *T* be the table identified by the <table name>.

2) The schema identified by the explicit or implicit schema name of the <table name> shall include the descriptor of *T*.

3) The scope of the <table name> is the entire <alter table statement>.

4) *T* shall be a base table.

5) *T* shall not be a declared local temporary table.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the <schema name> of the table identified by <table name>.

General Rules

1) The base table descriptor of *T* is modified as specified by <alter table action>.

2) If <add column definition> or <drop column definition> is specified, then the row type *RT* of *T* is the set of pairs (<field name>, <data type>) where <field name> is the name of a column *C* of *T* and <data type> is the declared type of *C*. This set of pairs contains one pair for each column of *T* in the order of their ordinal position in *T*.

Conformance Rules

None.

11.11 <add column definition>

Function

Add a column to a table.

Format

```
<add column definition> ::=
  ADD [ COLUMN ] <column definition>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> immediately contained in the containing <alter table statement>.

2) *T* shall not be a referenceable table.

3) If <column definition> contains <identity column specification>, then the table descriptor of *T* shall not include a column descriptor of an identity column.

Access Rules

None.

General Rules

1) The column defined by the <column definition> is added to *T*.

2) Let *C* be the column added to *T*.

   Case:

   a) If *C* is a generated column, then let *TN* be the <table name> immediately contained in the containing <alter table statement>, let *CN* be the <column name> immediately contained in <column definition>, and let *GE* be the generation expression included in the column descriptor of *C*. The following <update statement: searched> is executed without further Syntax Rule or Access Rule checking:

      ```
      UPDATE TN SET CN = GE
      ```

   b) Otherwise, *C* is a base column.

      Case:

      i) If *C* is an identity column, then for each row in *T* let *CS* be the site corresponding to *C* and let *NV* be the result of applying the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator", with the sequence descriptor included in the column descriptor of *C* as *SEQUENCE*.

         Case:

1) If the declared type of *C* is a distinct type *DIST*, then let *CNV* be *DIST*(*NV*).

2) Otherwise, let *CNV* be *NV*.

The General Rules of Subclause 9.2, "Store assignment", are applied with *CS* as *TARGET* and *CNV* as *VALUE*.

ii) Otherwise, every value in *C* is the default value for *C*.

NOTE 288 — The default value of a column is defined in Subclause 11.5, "<default clause>".

NOTE 289 — The addition of a column to a table has no effect on any existing <query expression> included in a view descriptor, <triggered action> included in a trigger descriptor, or <search condition> included in a constraint descriptor because any implicit column references in these descriptor elements are syntactically substituted by explicit column references under the Syntax Rules of Subclause 7.12, "<query specification>". Furthermore, by implication (from the lack of any General Rules to the contrary), the meaning of a column reference is never retroactively changed by the addition of a column subsequent to the invocation of the <SQL schema statement> containing that column reference.

3) For every table privilege descriptor that specifies *T* and a privilege of SELECT, UPDATE, INSERT or REFERENCES, a new column privilege descriptor is created that specifies *T*, the same action, grantor, and grantee, and the same grantability, and specifies the <column name> of the <column definition>.

4) In all other respects, the specification of a <column definition> in an <alter table statement> has the same effect as specification of the <column definition> in the <table definition> for *T* would have had. In particular, the degree of *T* is increased by 1 (one) and the ordinal position of that column is equal to the new degree of *T* as specified in the General Rules of Subclause 11.4, "<column definition>".

Conformance Rules

None.

11.12 <alter column definition>

Function

Change a column and its definition.

Format

```
<alter column definition> ::=

  ALTER [ COLUMN ] <column name> <alter column action>

<alter column action> ::=
    <set column default clause>
  | <drop column default clause>
  | <add column scope clause>
  | <drop column scope clause>
  | <alter identity column specification>
```

Syntax Rules

1) Let *T* be the table identified in the containing <alter table statement>.

2) Let *C* be the column identified by the <column name>.

3) *C* shall be a column of *T*.

4) If *C* is the self-referencing column of *T* or *C* is a generated column of *T*, then <alter column action> shall not contain <add column scope clause> or <drop column scope clause>.

5) If *C* is an identity column, then <alter column action> shall contain <alter identity column specification>.

6) If <alter identity column specification> is specified, then *C* shall be an identity column.

Access Rules

None.

General Rules

1) The column descriptor of *C* is modified as specified by <alter column action>.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <alter column definition>.

11.13 <set column default clause>

Function

Set the default clause for a column.

Format

```
<set column default clause> ::=
  SET <default clause>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>.

2) The default value specified by the <default clause> is placed in the column descriptor of *C*.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <set column default clause>.

11.14 <drop column default clause>

Function

Drop the default clause from a column.

Format

```
<drop column default clause> ::=
  DROP DEFAULT
```

Syntax Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>.

2) The descriptor of *C* shall include a default value.

Access Rules

None.

General Rules

1) The default value is removed from the column descriptor of *C*.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop column default clause>.

11.15 <add column scope clause>

Function

Add a non-empty scope for an existing column of data type REF in a base table.

Format

```
<add column scope clause> ::=
  ADD <scope clause>
```

Syntax Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>. The declared type of *C* shall be some reference type. Let *RTD* be the reference type descriptor included in the descriptor of *C*.

2) Let *T* be the table identified by the <table name> in the containing <alter table statement>. If *T* is a referenceable table, then *C* shall be an originally-defined column of *T*.

3) *RTD* shall not include a scope.

4) Let *UDTN* be the name of the referenced type included in *RTD*.

5) The <table name> *STN* contained in the <scope clause> shall identify a referenceable table whose structured type is *UDTN*.

Access Rules

None.

General Rules

1) *STN* is included as the scope in the reference type descriptor included in the column descriptor of *C*.

2) For any proper subtable *PST* of *T*, let *PSC* be the column whose corresponding column in *T* is *C*. *STN* is included as the scope in the reference type descriptor included in the column descriptor of *PSC*.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <add column scope clause>.

2) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain an <add column scope clause>.

11.16 <drop column scope clause>

This Subclause is modified by Subclause 9.7, "<drop column scope clause>", in ISO/IEC 9075-4.

Function

Drop the scope from an existing column of data type REF in a base table.

Format

```
<drop column scope clause> ::=
  DROP SCOPE <drop behavior>
```

Syntax Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>. The declared type of *C* shall be some reference type whose reference type descriptor includes a scope.

2) Let *T* be the table identified by the <table name> in the containing <alter table statement>. If *T* is a referenceable table, then *C* shall be an originally-defined column of *T*.

3) Let *SC* be the set of columns consisting of *C* and, for every proper subtable of *T*, the column whose supercolumn is *C*.

4) An *impacted dereference operation* is a <dereference operation> whose <reference value expression> is a column reference that identifies a column in *SC*, a <method reference> whose <value expression primary> is a column reference that identifies a column in *SC*, or a <reference resolution> whose <reference value expression> is a column reference that identifies a column in *SC*.

5) If RESTRICT is specified, then no impacted dereference operation shall be contained in any of the following:

   a) The SQL routine body of any routine descriptor.

   b) The <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) The triggered action of any trigger descriptor.

   NOTE 290 — If CASCADE is specified, then such referencing objects will be dropped by the execution of the <SQL procedure statement>s specified in the General Rules of this Subclause.

Access Rules

None.

General Rules

1) For every SQL-invoked routine *R* whose routine descriptor includes an SQL routine body that contains an impacted dereference operation, let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed for every *R* without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

2) For every view *V* whose view descriptor includes a <query expression> that contains an impacted dereference operation, let *VN* be the <table name> of *V*. The following <drop view statement> is effectively executed for every *V* without further Access Rule checking:

```
DROP VIEW VN CASCADE
```

3) For every assertion *A* whose assertion descriptor includes a <search condition> that contains an impacted dereference operation, let *AN* be the <constraint name> of *A*. The following <drop assertion statement> is effectively executed for every *A* without further Access Rule checking:

```
DROP ASSERTION AN CASCADE
```

4) For every table check constraint *CC* whose table check constraint descriptor includes a <search condition> that contains an impacted dereference operation, let *CN* be the <constraint name> of *CC* and let *TN* be the <table name> of the table whose descriptor includes descriptor of *CC*. The following <alter table statement> is effectively executed for every *CC* without further Access Rule checking:

```
ALTER TABLE TN DROP CONSTRAINT CN CASCADE
```

5) The scope included in the reference type descriptor included in the column descriptor of every column in *SC* is made empty.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop column scope clause>.

2) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <drop column scope clause>.

11.17 <alter identity column specification>

Function

Change the options specified for an identity column.

Format

```
<alter identity column specification> ::=
  <alter identity column option>...

<alter identity column option> ::=
    <alter sequence generator restart option>
  | SET <basic sequence generator option>
```

Syntax Rules

1) Let *SEQ* be the sequence generator descriptor included in the column descriptor identified by the <column name> in the containing <alter column definition>.

2) Let *OPT* be the character string formulated from <alter identity column specification> that conforms to the Format of <alter sequence generator options>.

   NOTE 291 — *OPT* is formulated by removing all instances of the keyword SET from the string corresponding to from <alter identity column specification>.

3) The Syntax Rules of Subclause 9.25, "Altering a sequence generator", are applied with *OPT* as *OPTIONS* and *SEQ* as *SEQUENCE*.

Access Rules

None.

General Rules

1) The General Rules of Subclause 9.25, "Altering a sequence generator", are applied with *OPT* as *OPTIONS* and *SEQ* as *SEQUENCE*.

Conformance Rules

1) Without Feature T174, "Identity columns", an <alter column definition> shall not contain an <alter identity column specification>.

11.18 <drop column definition>

This Subclause is modified by Subclause 9.8, "<drop column definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.6, "<drop column definition>", in ISO/IEC 9075-9.

Function

Destroy a column of a base table.

Format

```
<drop column definition> ::=
  DROP [ COLUMN ] <column name> <drop behavior>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> in the containing <alter table statement> and let *TN* be the name of *T*.

2) Let *C* be the column identified by the <column name> *CN*.

3) *T* shall not be a referenceable table.

4) *C* shall be a column of *T* and *C* shall not be the only column of *T*.

5) If RESTRICT is specified, then *C* shall not be referenced in any of the following:

   a) The <query expression> of any view descriptor.

   b) The <search condition> of any constraint descriptor other than a table constraint descriptor that contains references to no other column and that is included in the table descriptor of *T*.

   c) The SQL routine body of any routine descriptor.

   d) Either an explicit trigger column list or a triggered action column set of any trigger descriptor.

   e) The generation expression of any column descriptor.

NOTE 292 — A <drop column definition> that does not specify CASCADE will fail if there are any references to that column resulting from the use of CORRESPONDING, NATURAL, SELECT * (except where contained in an <exists predicate>), a multi-column <unique constraint definition>, or REFERENCES without a <reference column list> in its <referenced table and columns>. Also note that a <drop column definition> whose <drop behavior> is RESTRICT that references a column that is referenced by a multi-column <unique constraint definition> will fail, while dropping a column that is referenced by a single-column <unique constraint definition> will succeed.

NOTE 293 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the <revoke statement> specified in the General Rules of this Subclause.

Access Rules

None.

General Rules

1) Let *TR* be the trigger name of any trigger descriptor having an explicit trigger column list or a triggered action column set that contains *CN*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER TR
```

2) Let *A* be the <authorization identifier> that owns *T*. The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE INSERT(CN), UPDATE(CN),
SELECT(CN),
REFERENCES(CN) ON TABLE
TN FROM A CASCADE
```

3) Let *GC* be any generated column of *T* in whose descriptor the generation expression contains a <column reference> that references *C*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE T DROP COLUMN GC CASCADE
```

4) If the column is not based on a domain, then its data type descriptor is destroyed; otherwise, all domain constraint usages that reference *C* are destroyed.

5) The data associated with *C* is destroyed.

6) The descriptor of *C* is removed from the descriptor of *T*.

7) The descriptor of *C* is destroyed.

8) The degree of *T* is reduced by 1 (one). The ordinal position of all columns having an ordinal position greater than the ordinal position of *C* is reduced by 1 (one).

Conformance Rules

1) Without Feature F033, "ALTER TABLE statement: DROP COLUMN clause", conforming SQL language shall not contain a <drop column definition>.

11.19 <add table constraint definition>

Function

Add a constraint to a table.

Format

```
<add table constraint definition> ::=
  ADD <table constraint definition>
```

Syntax Rules

1) If PRIMARY KEY is specified, then *T* shall not have any proper supertable.

Access Rules

None.

General Rules

1) Let *T* be the table identified by the <table name> in the containing <alter table statement>.

2) The table constraint descriptor for the <table constraint definition> is included in the table descriptor for *T*.

3) Let *TC* be the table constraint added to *T*. If *TC* causes some column *CN* to be known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of the column descriptor of *CN* is changed to known not nullable.

   NOTE 294 — The nullability characteristic of a column is defined in Subclause 4.13, “Columns, fields, and attributes”.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <add table constraint definition>.

11.20 <drop table constraint definition>

This Subclause is modified by Subclause 9.9, "<drop table constraint definition>", in ISO/IEC 9075-4.

Function

Destroy a constraint on a table.

Format

```
<drop table constraint definition> ::=
  DROP CONSTRAINT <constraint name> <drop behavior>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> in the containing <alter table statement>.

2) The <constraint name> shall identify a table constraint *TC* of *T*.

3) If *TC* is a unique constraint and *RC* is a referential constraint whose referenced table is *T* and whose referenced columns are the unique columns of *TC*, then *RC* is said to be *dependent on TC*.

4) If *QS* is a <query specification> that contains an implicit or explicit <group by clause> and that contains a column reference to a column *C* in its <select list> that is not contained in an aggregated argument of a <set function specification>, and if *G* is the set of grouping columns of *QS*, and if the table constraint *TC* is needed to conclude that *G* *C* is a known functional dependency in *QS*, then *QS* is said to be *dependent on TC*.

5) If *V* is a view that contains a <query specification> that is dependent on a table constraint *TC*, then *V* is said to be *dependent on TC*.

6) If *R* is an SQL routine whose <SQL routine body> contains a <query specification> that is dependent on a table constraint *TC*, then *R* is said to be *dependent on TC*.

7) If *C* is a constraint or assertion whose <search condition> contains a <query specification> that is dependent on a table constraint *TC*, then *C* is said to be *dependent on TC*.

8) If *T* is a trigger whose triggered action contains a <query specification> that is dependent on a table constraint *TC*, then *T* is said to be *dependent on TC*.

9) If *T* is a referenceable table with a derived self-referencing column, then:

   a) *TC* shall not be a unique constraint whose unique columns correspond to the attributes in the list of attributes of the derived representation of the reference type whose referenced type is the structured type of *T*.

   b) *TC* shall not be a unique constraint whose unique column is the self-referencing column of *T*.

10) If RESTRICT is specified, then:

   a) No table constraint shall be dependent on *TC*.

b) The <constraint name> of *TC* shall not be generally contained in the SQL routine body of any routine descriptor.

c) No view shall be dependent on *TC*.

d) No SQL routine shall be dependent on *TC*.

e) No constraint or assertion shall be dependent on *TC*.

f) No trigger shall be dependent on *TC*.

NOTE 295 — If CASCADE is specified, then any such dependent object will be dropped by the effective execution of the General Rules of this Subclause.

Access Rules

None.

General Rules

1) Let *TCN2* be the <constraint name> of any table constraint that is dependent on *TC* and let *T2* be the <table name> of the table descriptor that includes *TCN2*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE T2 DROP
CONSTRAINT TCN2 CASCADE
```

2) Let *R* be any SQL-invoked routine whose routine descriptor contains the <constraint name> of *TC* in the SQL routine body. Let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

3) Let *VN* be the table name of any view *V* that is dependent on *TC*. The following <drop view statement> is effectively executed for every *V*:

```
DROP VIEW VN CASCADE
```

4) Let *SN* be the specific name of any SQL routine *R* that is dependent on *TC*. The following <drop routine statement> is effectively executed for every *SR*:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

5) Let *CN* be the constraint name of any constraint *C* that is dependent on *TC*. Let *TN* be the name of the table constrainted by *C*. The following <alter table statement> is effectively executed for every *C*:

```
ALTER TABLE TN DROP
CONSTRAINT CN CASCADE
```

6) Let *AN* be the assertion name of any assertion *A* that is dependent on *TC*. The following <drop assertion statement> is effectively executed for every *A*:

```
DROP ASSERTION AN CASCADE
```

7) Let *TN* be the trigger name of any trigger *T* that is dependent on *TC*. The following <drop trigger statement> is effectively executed for every *T*:

```
DROP TRIGGER T CASCADE
```

8) The descriptor of *TC* is removed from the descriptor of *T*.

9) If *TC* causes some column *CN* to be known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of the column descriptor of *CN* is changed to possibly nullable.

   NOTE 296 — The nullability characteristic of a column is defined in Subclause 4.13, "Columns, fields, and attributes".

10) The descriptor of *TC* is destroyed.

Conformance Rules

1) Without Feature F381, "Extended schema manipulation", conforming SQL language shall not contain a <drop table constraint definition>.

11.21 <drop table statement>

This Subclause is modified by Subclause 9.10, "<drop table statement>", in ISO/IEC 9075-4.

Function

Destroy a table.

Format

```
<drop table statement> ::=
  DROP TABLE <table name> <drop behavior>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> and let *TN* be that <table name>.

2) The schema identified by the explicit or implicit schema name of the <table name> shall include the descriptor of *T*.

3) *T* shall be a base table.

4) *T* shall not be a declared local temporary table.

5) An *impacted dereference operation* is any of the following:

   a) A <dereference operation> *DO*, where *T* is the scope of the reference type of the <reference value expression> immediately contained in *DO*.

   b) A <method reference> *MR*, where *T* is the scope of the reference type of the <value expression primary> immediately contained in *MR*.

   c) A <reference resolution> *RR*, where *T* is the scope of the reference type of the <reference value expression> immediately contained in *RR*.

6) If RESTRICT is specified, then *T* shall not have any proper subtables.

7) If RESTRICT is specified, then *T* shall not be referenced and no impacted dereference operation shall be contained in any of the following:

   a) The <query expression> of any view descriptor.

   b) The <search condition> of any constraint descriptor that is not a table check constraint descriptor included in the base table descriptor of *T*.

   c) The <search condition> of any assertion descriptor.

   d) The table descriptor of the referenced table of any referential constraint descriptor of any table other than *T*.

   e) The SQL routine body of any routine descriptor.

   f) The <triggered action> of any trigger descriptor.

NOTE 297 — If CASCADE is specified, then such objects will be dropped by the execution of the <revoke statement> specified in the General Rules of this Subclause.

8) If RESTRICT is specified and *T* is a referenceable table, then *TN* shall not be the scope included in a reference type descriptor generally included in any of the following:

   a) The attribute descriptor of an attribute of a user-defined type.

   b) The column descriptor of a column of a table other than *T*.

   c) The descriptor of an SQL parameter or the result type included in the routine descriptor of any <SQL-invoked routine>.

   d) The descriptor of an SQL parameter or the result type included in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

   e) The descriptor of any user-defined cast.

   NOTE 298 — A descriptor that "generally includes" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

9) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the table identified by *TN*.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) Let *STN* be the <table name> of any direct subtable of *T*. The following <drop table statement> is effectively executed without further Access Rule checking:

```
DROP TABLE STN CASCADE
```

2) For every proper supertable of *T*, every superrow of every row of *T* is effectively deleted at the end of the SQL-statement, prior to the checking of any integrity constraints.

   NOTE 299 — This deletion creates neither a new trigger execution context nor the definition of a new state change in the current trigger execution context.

3) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE ALL PRIVILEGES ON TN FROM A CASCADE
```

4) If *T* is a referenceable table, then:

   a) For every reference type descriptor *RTD* that includes a scope of *TN* and is generally included in any of the following:

      i) The attribute descriptor of an attribute of a user-defined type.

      ii) The column descriptor of a column of a table other than *T*.

iii) The descriptor of an SQL parameter or the result type included in the routine descriptor of any <SQL-invoked routine>.

iv) The descriptor of an SQL parameter or the result type in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

v) The descriptor of any user-defined cast.

the scope of *RTD* is made empty.

b) Let *SOD* be the descriptor of a schema object dependent on the table descriptor of *T*.

Case:

i) If *SOD* is a view descriptor, then let *SON* be the name of the view included in *SOD*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SON CASCADE
```

ii) If *SOD* is an assertion descriptor, then let *SON* be the name of the assertion included in *SOD*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION SON CASCADE
```

iii) If *SOD* is a table constraint descriptor, then let *SON* be the name of the constraint included in *SOD*. Let *CTN* be the <table name> included in the table descriptor that includes *SOD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE CTN DROP CONSTRAINT SON CASCADE
```

iv) If *SOD* is a routine descriptor, then let *SON* be the specific name included in *SOD*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

v) If *SOD* is a trigger descriptor, then let *SON* be the trigger name included in *SOD*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER SON CASCADE
```

NOTE 300 — A descriptor that "depends on" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

5) For each direct supertable *DST* of *T*, the table name of *T* is removed from the list of table names of direct subtables of *DST* that is included in the table descriptor of *DST*.

6) The descriptor of *T* is destroyed.

Conformance Rules

1) Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop table statement> that contains <drop behavior> that contains CASCADE.

11.22 <view definition>

This Subclause is modified by Subclause 9.11, "<view definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 9.3, "<view definition>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.3, "<view definition>", in ISO/IEC 9075-14.

Function

Define a viewed table.

Format

```
<view definition> ::=
  CREATE [ RECURSIVE ] VIEW <table name> <view specification>
      AS <query expression> [ WITH [ <levels clause> ] CHECK OPTION ]

<view specification> ::=
    <regular view specification>
  | <referenceable view specification>

<regular view specification> ::=
  [ <left paren> <view column list> <right paren> ]

<referenceable view specification> ::=
  OF <path-resolved user-defined type name> [ <subview clause> ]
      [ <view element list> ]

<subview clause> ::=
  UNDER <table name>

<view element list> ::=
  <left paren> <view element> [ { <comma> <view element> }... ] <right paren>

<view element> ::=
    <self-referencing column specification>
  | <view column option>

<view column option> ::=
  <column name> WITH OPTIONS <scope clause>

<levels clause> ::=
    CASCADED
  | LOCAL

<view column list> ::=
  <column name list>
```

Syntax Rules

1) The <query expression> shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, or an <embedded variable specification>.

2) If a <view definition> is contained in a <schema definition> and the <table name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

3) The schema identified by the explicit or implicit schema name of the <table name> shall not include a table descriptor whose table name is <table name>.

4) No <table reference> generally contained in the <query expression> shall identify any declared local temporary table.

5) If a <table reference> generally contained in the <query expression> identifies the viewed table *VT* defined by <view definition> *VD*, then *VD* and *VT* are said to be *recursive*.

6) If *VD* is recursive, then:

   a) <view column list> shall be specified.

   b) RECURSIVE shall be specified.

   c) CHECK OPTION shall not be specified.

   d) <referenceable view specification> shall not be specified.

   e) *VD* is equivalent to

```
CREATE VIEW <table name> AS
    WITH RECURSIVE <table name> (<view column list>)
      AS (<query expression>)
    SELECT <view column list> FROM <table name>
```

7) The viewed table is *updatable* if the <query expression> is updatable.

8) The viewed table is *simply updatable* if the <query expression> is simply updatable.

9) The viewed table is *effectively updatable* if it is simply updatable, or if the SQL implementation supports Feature T111, “Updatable joins, unions, and columns”, and the viewed table is updatable.

10) The viewed table is *insertable-into* if the <query expression> is insertable-into.

11) If any two columns in the table specified by the <query expression> have equivalent <column name>s, or if any column of that table has an implementation-dependent name, then a <view column list> shall be specified.

12) Equivalent <column name>s shall not be specified more than once in the <view column list>.

13) The number of <column name>s in the <view column list> shall be the same as the degree of the table specified by the <query expression>.

14) Every column in the table specified by <query expression> whose declared type is a character string type shall have a declared type collation.

15) If WITH CHECK OPTION is specified, then the viewed table shall be updatable.

16) If WITH CHECK OPTION is specified and <levels clause> is not specified, then a <levels clause> of CASCADED is implicit.

17) If WITH LOCAL CHECK OPTION is specified, then the <query expression> shall not generally contain a <query expression> *QE* or a <query specification> *QS* that is *possibly non-deterministic* unless *QE* or *QS* is generally contained in a viewed table that is a leaf underlying table of the <query expression>.

    If WITH CASCADED CHECK OPTION is specified, then the <query expression> shall not generally contain a <query expression> or <query specification> that is *possibly non-deterministic*.

18) Let *V* be the view defined by the <view definition>. The underlying columns of every *i*-th column of *V* are the underlying columns of the *i*-th column of the <query expression> and the underlying columns of *V* are the underlying columns of the <query expression>.

19) <subview clause>, if present, identifies the *direct superview SV* of *V* and *V* is said to be a *direct subview* of *SV*. View *V1* is a *superview* of view *V2* if and only if one of the following is true:

    a) *V1* and *V2* are the same view.

    b) *V1* is a direct superview of *V2*.

    c) There exists a view *V3* such that *V1* is a direct superview of *V3* and *V3* is a superview of *V2*. If *V1* is a superview of *V2*, then *V2* is a subview of *V1*.

    If *V1* is a superview of *V2* and *V1* and *V2* are not the same view, then *V2* is a *proper subview* of *V1* and *V1* is a *proper superview* of *V2*.

    If *V2* is a direct subview of *V1*, then *V2* is a direct subtable of *V1*.

    NOTE 301 — It follows that the subviews of the superviews of *V* together constitute the subtable family of *V*, every implication of which applies.

20) If <referenceable view specification> is specified, then:

    a) *V* is a *referenceable view*.

    b) RECURSIVE shall not be specified.

    c) The <user-defined type name> simply contained in <path-resolved user-defined type name> shall identify a structured type *ST*.

    d) The subtable family of *V* shall not include a member, other than *V* itself, whose associated structured type is *ST*.

    e) If <subview clause> is not specified, then <self-referencing column specification> shall be specified.

    f) Let *QE* be the <query expression>.

    g) Let *n* be the number of attributes of *ST*. Let A_i, 1 (one) $\leq i \leq n$ be the attributes of *ST*.

    h) Let *RT* be the row type of *QE*.

    i) If <self-referencing column specification> is specified, then:

        i) Exactly one <self-referencing column specification> shall be specified.

        ii) <subview clause> shall not be specified.

        iii) SYSTEM GENERATED shall not be specified.

        iv) Let *RST* be the reference type REF(*ST*).

Case:

1) If USER GENERATED is specified, then:

   A) *RST* shall have a user-defined representation.

   B) Let *m* be 1 (one).

2) If DERIVED is specified, then:

   A) *RST* shall have a derived representation.

   B) Let *m* be 0 (zero).

j) If <subview clause> is specified, then:

   i) The <table name> contained in the <subview clause> shall identify a referenceable table *SV* that is a view.

   ii) *ST* shall be a direct subtype of the structured type of the direct supertable of *V*.

   iii) The SQL-schema identified by the explicit or implicit <schema name> of the <table name> of *V* shall include the descriptor of *SV*.

   iv) Let *MSV* be the maximum superview of the subtable family of *V*. Let *RMSV* be the reference type REF(*MSV*).

   Case:

   1) If *RMSV* has a user-defined representation, then let *m* be 1 (one).

   2) Otherwise, *RMSV* has a derived representation. Let *m* be 0 (zero).

k) The degree of *RT* shall be $n+m$.

l) Let F_i, 1 (one) $\leq i \leq n$, be the fields of *RT*.

m) For *i* varying from 1 (one) to *n*:

   i) The declared data type $DDTF_{i+m}$ of F_{i+m} shall be compatible with the declared data type $DDTA_i$ of A_i.

   ii) The Syntax Rules of Subclause 9.18, "Data type identity", are applied with $DDTF_{i+m}$ and $DDTA_i$.

n) *QE* shall consist of a single <query specification> *QS*.

o) The <from clause> of *QS* shall simply contain a single <table reference> *TR*.

p) *TR* shall immediately contain a <table or query name>. Let *TQN* be the table identified by the <table or query name>. *TQN* is the *basis table* of *V*.

q) If *TQN* is a referenceable base table or a referenceable view, then *TR* shall simply contain ONLY.

r) *QS* shall not simply contain a <group by clause> or a <having clause>.

s) If <self-referencing column specification> is specified, then

   Case:

i) If *RST* has a user-defined representation, then:

1) *TQN* shall have a candidate key consisting of a single column *RC*.

2) Let *SS* be the first <select sublist> in the <select list> of *QS*.

3) *SS* shall consist of a single <cast specification> *CS* whose leaf column is *RC*.

NOTE 302 — "Leaf column of a <cast specification>" is defined in Subclause 6.12, "<cast specification>".

4) The declared type of F_1 shall be REF(*ST*).

ii) Otherwise, *RST* has a derived representation.

1) Let C_i, 1 (one) ≤ i ≤ n, be the columns of *V* that correspond to the attributes of the derived representation of *RST*.

2) *TQN* shall have a candidate key consisting of some subset of the underlying columns of C_i, 1 (one) ≤ i ≤ n.

t) If <subview clause> is specified, then *TQN* shall be a proper subtable or proper subview of the basis table of *SV*.

u) Let <view element list>, if specified, be *TEL1*.

v) Let r be the number of <view column option>s. For every <view column option> VCO_j, 1 (one) ≤ j ≤ r, <column name> shall be equivalent to the <attribute name> of some attribute of *ST*.

w) Distinct <view column option>s contained in *TEL1* shall specify distinct <column name>s.

x) Let CN_j, 1 (one) ≤ j ≤ r, be the <column name> contained in VCO_j and let SCL_j be the <scope clause> contained in VCO_j.

i) CN_j shall be equivalent to some <attribute name> of *ST*, whose declared type is some reference type $CORT_j$.

ii) The <table name> contained in SCL_j shall identify a referenceable table *SRT*.

iii) *SRT* shall be based on the referenced type of $CORT_j$.

NOTE 303 — The notion of one data type being based on another data type is defined in Subclause 4.1, "Data types".

21) Let the *originally-defined columns* of *V* be the columns of the table defined by *QE*.

22) A column of *V* is called an *updatable column* of *V* if its underlying column is updatable.

23) If the <view definition> is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>; otherwise, let *A* be the <authorization identifier> that owns the schema identified by the explicit or implicit <schema name> of the <table name>.

Access Rules

1) If a <view definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <table name>.

2) If <referenceable view specification> is specified, then the applicable privileges for *A* shall include USAGE on *ST*.

3) If <subview clause> is specified, then

   Case:

   a) If <view definition> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the schema shall include UNDER for *SV*.

   b) Otherwise, the current privileges shall include UNDER for *SV*.

General Rules

1) A view descriptor *VD* is created that describes *V*. *VD* includes:

   a) The <table name> *TN*.

   b) *QE*, as both the <query expression> of the descriptor and the original <query expression> of the descriptor.

   c) Case:

      i) If <regular view specification> is specified, then the column descriptors taken from the table specified by the <query expression>.

         Case:

         1) If a <view column list> is specified, then the <column name> of the *i*-th column of the view is the *i*-th <column name> in that <view column list>.

         2) Otherwise, the <column name>s of the view are the <column name>s of the table specified by the <query expression>.

      ii) Otherwise:

         1) A column descriptor in which:

            A) The name of the column is <self-referencing column name>.

            B) The data type descriptor is that generated by the <data type> “REF(*ST*) SCOPE(*TN*)”.

            C) The nullability characteristic is *known not nullable*.

            D) The ordinal position is 1 (one).

            E) The column is indicated to be self-referencing.

         2) The column descriptor *ODCD* of each originally-defined column *ODC* of *V* in which:

            A) The <column name> included in *ODCD* is replaced by the <attribute name> of its corresponding attribute of *ST*.

            B) If the declared type of the column is a reference type and some VCO_i contains the <attribute name> of *ST* that corresponds to the column, then the (possibly empty) scope

contained in the reference type descriptor immediately included in the column descriptor is replaced by SCO_i.

3) If DERIVED is specified, then an indication that the self-referencing column is a derived self-referencing column.

4) If USER GENERATED is specified, then an indication that the self-referencing column is a user-generated self-referencing column.

d) In each column descriptor, an indication that the column is updatable if *V* is effectively updatable and the corresponding column of *QE* is updatable.

e) An indication as to whether WITH CHECK OPTION was omitted, specified with LOCAL, or specified with CASCADED.

2) Let *VN* be the <table name>. Let *QE* be the <query expression> included in the view descriptor *VD* of the view identified by *VN*. Let *OQE* be the original <query expression> included in *VD*. If a <view column list> is specified, then let *VCL* be the <view column list> preceded by a <left paren> and followed by a <right paren>; otherwise, let *VCL* be the zero-length string.

Case:

a) If *VN* is immediately contained in some SQL-schema statement, then *VN* identifies the view descriptor *VD*.

b) If *VN* is immediately contained in a <table reference> that specifies ONLY, then *VN* references the same table as the <table reference>:

```
( OQE ) AS VN VCL
```

c) Otherwise, *VN* references the same table as the <table reference>:

```
( QE ) AS VN VCL
```

3) For *i* ranging from 1 (one) to the number of distinct leaf underlying tables of the <query expression> *QE* of *V*, let RT_i be the <table name>s of those tables. For every column *CV* of *V*:

a) Let $CRT_{i,j}$, for *j* ranging from 1 (one) to the number of columns of RT_i that are underlying columns of *CV*, be the <column name>s of those columns.

b) A set of privilege descriptors with the grantor for each set to the special grantor value "_SYSTEM" is created as follows:

i) For every column *CV* of *V*, a privilege descriptor is created that defines the privilege SELECT(*CV*) on *V* to *A*. That privilege is grantable if and only if all the following are true:

1) The applicable privileges for *A* include grantable SELECT privileges on all of the columns $CRT_{i,j}$.

2) The applicable privileges for *A* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

3) The applicable privileges for *A* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

4) The applicable privileges for *A* include grantable SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

ii) For every column *CV* of *V*, if the applicable privileges for *A* include REFERENCES($CRT_{i,j}$) for all *i* and for all *j*, and the applicable privileges for *A* include REFERENCES on some column of RT_i for all *i*, then a privilege descriptor is created that defines the privilege REFERENCES(*CV*) on *V* to *A*. That privilege is grantable if and only if all the following are true:

1) The applicable privileges for *A* include grantable REFERENCES privileges on all of the columns $CRT_{i,j}$.

2) The applicable privileges for *A* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

3) The applicable privileges for *A* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

4) The applicable privileges for *A* include grantable SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

4) A privilege descriptor is created that defines the privilege SELECT on *V* to *A*. That privilege is grantable if and only if the applicable privileges for *A* include grantable SELECT privilege on every column of *V*. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

5) If the applicable privileges for *A* include REFERENCES privilege on every column of *V*, then a privilege descriptor is created that defines the privilege REFERENCES on *V* to *A*. That privilege is grantable if and only if the applicable privileges for *A* include grantable REFERENCES privilege on every column of *V*. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

6) If *V* is effectively updatable, then a set of privilege descriptors is created by applying the General Rules of Subclause 10.11, "Determination of view and view component privileges", with *V* as the *VIEW*.

7) If *V* is a referenceable view, then a set of privilege descriptors with the grantor for each set to the special grantor value "_SYSTEM" are created as follows:

a) A privilege descriptor is created that defines the SELECT privilege WITH HIERARCHY OPTION on *V* to *A*. That privilege is grantable.

b) For every method *M* of the structured type identified by <path-resolved user-defined type name>, a privilege descriptor is created that defines the privilege SELECT(*M*) on *V* to *A*. That privilege is grantable.

c) Case:

i) If <subview clause> is not specified, then a privilege descriptor is created that defines the UNDER privilege on *V* to *A*. That privilege is grantable.

ii) Otherwise, a privilege descriptor is created that defines the UNDER privilege on *V* to *A*. That privilege is grantable if and only if the applicable privileges for *A* include grantable UNDER privilege on the direct supertable of *V*.

8) If <subview clause> is specified, then let *ST* be the set of supertables of *V*. Let *PDS* be the set of privilege descriptors that define SELECT WITH HIERARCHY OPTION privilege on a table in *ST*.

9) For every privilege descriptor in *PDS*, with grantee *G* and grantor *A*,

   Case:

   a) If the privilege is grantable, then let *WGO* be "WITH GRANT OPTION".

   b) Otherwise, let *WGO* be a zero-length string.

   The following <grant statement> is effectively executed without further Access Rule checking:

   ```
   GRANT SELECT ON V TO G WGO FROM A
   ```

10) If <subview clause> is specified, then let *SVQE* be the <query expression> included in the view descriptor of *SV*.

    a) The <query expression> included in the descriptor of *SV* is replaced by the following <query expression>:

       ```
       ( SVQE ) UNION ALL CORRESPONDING SELECT * FROM TN
       ```

    b) The General Rules of this subclause are reevaluated for *SV* in the light of the new <query expression> in its descriptor.

11) If the character representation of the <query expression> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — query expression too long for information schema*.

    NOTE 304 — The Information Schema is defined in ISO/IEC 9075-11.

Conformance Rules

1) Without Feature T131, "Recursive query", conforming SQL language shall not contain a <view definition> that immediately contains RECURSIVE.

2) Without Feature F751, "View CHECK enhancements", conforming SQL language shall not contain a <levels clause>.

3) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <referenceable view specification>.

4) Without Feature F751, "View CHECK enhancements", conforming SQL language shall not contain <view definition> that contains a <query expression> that contains a <query expression> and contains WITH CHECK OPTION.

5) Without Feature T111, "Updatable joins, unions and columns", in conforming SQL language, if WITH CHECK OPTION is specified, then the viewed table shall be simply updatable.

11.23 <drop view statement>

This Subclause is modified by Subclause 9.12, "<drop view statement>", in ISO/IEC 9075-4.

Function

Destroy a view.

Format

```
<drop view statement> ::=
  DROP VIEW <table name> <drop behavior>
```

Syntax Rules

1) Let *V* be the table identified by the <table name> and let *VN* be that <table name>. The schema identified by the explicit or implicit schema name of *VN* shall include the descriptor of *V*.

2) *V* shall be a viewed table.

3) An *impacted dereference operation* is any of the following:

   a) A <dereference operation> *DO*, where *V* is the scope of the reference type of the <reference value expression> immediately contained in *DO*.

   b) A <method reference> *MR*, where *V* is the scope of the reference type of the <value expression primary> immediately contained in *MR*.

   c) A <reference resolution> *RR*, where *V* is the scope of the reference type of the <reference value expression> immediately contained in *RR*.

4) If RESTRICT is specified, then *V* shall not have any proper subtables.

5) If RESTRICT is specified, then *V* shall not be referenced and no impacted dereference operation shall be contained in any of the following:

   a) The <query expression> of the view descriptor of any view other than *V*.

   b) The <search condition> of any constraint descriptor or assertion descriptor.

   c) The <triggered action> of any trigger descriptor.

   d) The SQL routine body of any routine descriptor.

   NOTE 305 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the <revoke statement> specified in the General Rules of this Subclause.

6) If RESTRICT is specified and *V* is a referenceable view, then *VN* shall not be the scope included in a reference type descriptor generally included in any of the following:

   a) The attribute descriptor of an attribute of a user-defined type.

   b) The column descriptor of a column of a table other than *V*.

c) The descriptor of an SQL parameter or the result type included in the routine descriptor of any <SQL-invoked routine>.

d) The descriptor of an SQL parameter or the result type included in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

e) The descriptor of any user-defined cast.

NOTE 306 — A descriptor that "generally includes" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

7) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the table identified by *VN*.

Access Rules

1) The enabled authorization identifier shall include *A*.

General Rules

1) Let *SVN* be the <table name> of any direct subview of *V*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SVN CASCADE
```

2) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE ALL PRIVILEGES ON VN FROM A CASCADE
```

3) If *V* is a referenceable view, then:

a) For every reference type descriptor *RTD* that includes a scope of *VN* and is generally included in any of the following:

i) The attribute descriptor of an attribute of a user-defined type.

ii) The column descriptor of a column of a table other than *V*.

iii) The descriptor of an SQL parameter or the result type included in the routine descriptor of any <SQL-invoked routine>.

iv) The descriptor of an SQL parameter or the result type included in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

v) The descriptor of any user-defined cast.

the scope of *RTD* is made empty.

b) Let *SOD* be the descriptor of a schema object dependent on the view descriptor of *V*.

Case:

i) If *SOD* is a view descriptor, then let *SON* be the name of the view included in *SOD*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SON CASCADE
```

ii) If *SOD* is an assertion descriptor, then let *SON* be the name of the assertion included in *SOD*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION SON CASCADE
```

iii) If *SOD* is a table constraint descriptor, then let *SON* be the name of the constraint included in *SOD*. Let *CTN* be the <table name> included in the table descriptor that includes *SOD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE CTN DROP CONSTRAINT SON CASCADE
```

iv) If *SOD* is a routine descriptor, then let *SON* be the specific name included in *SOD*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

v) If *SOD* is a trigger descriptor, then let *SON* be the trigger name included in *SOD*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER SON
```

NOTE 307 — A descriptor that "depends on" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

4) For each direct supertable *DST* of *V*, the table name of *V* is removed from the list of table names of direct subtables of *DST* that is included in the table descriptor of *DST*.

5) The descriptor of *V* is destroyed.

Conformance Rules

1) Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop view statement> that contains a <drop behavior> that contains CASCADE.

11.24 <domain definition>

This Subclause is modified by Subclause 11.7, "<domain definition>", in ISO/IEC 9075-9.

Function

Define a domain.

Format

```
<domain definition> ::=
  CREATE DOMAIN <domain name> [ AS ] <predefined type>
      [ <default clause> ]
      [ <domain constraint>... ]
      [ <collate clause> ]

<domain constraint> ::=
  [ <constraint name definition> ] <check constraint definition> [
      <constraint characteristics> ]
```

Syntax Rules

1) If a <domain definition> is contained in a <schema definition>, and if the <domain name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

2) If <constraint name definition> is specified and its <constraint name> contains a <schema name>, then that <schema name> shall be equivalent to the explicit or implicit <schema name> of the <domain name> of the domain identified by the containing <domain definition> or <alter domain statement>.

3) The schema identified by the explicit or implicit schema name of the <domain name> shall not include a domain descriptor whose domain name is equivalent to <domain name> nor a user-defined type descriptor whose user-defined type name is equivalent to <domain name>.

4) If <predefined type> specifies a <character string type> and does not specify <character set specification>, then the character set name of the default character set of the schema identified by the implicit or explicit <schema name> of <domain name> is implicit.

5) <collate clause> shall not be both specified in <predefined type> and immediately contained in <domain definition>. If <collate clause> is immediately contained in <domain definition>, then it is equivalent to specifying an equivalent <collate clause> in <predefined type>.

6) Let *D1* be some domain. *D1* is *in usage by* a domain constraint *DC* if and only if the <search condition> of *DC* generally contains the <domain name> either of *D1* or of some domain *D2* such that *D1* is in usage by some domain constraint of *D2*. No domain shall be in usage by any of its own constraints.

7) If <collate clause> is specified, then <predefined type> shall be a character string type.

8) For every <domain constraint> specified:

   a) If <constraint characteristics> is not specified, then INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

b) If <constraint name definition> is not specified, then a <constraint name definition> that contains an implementation-dependent <constraint name> is implicit. The assigned <constraint name> shall obey the Syntax Rules of an explicit <constraint name>.

c) Let *S* be the schema identified by the explicit or implicit <schema name> of the <constraint name> contained in <domain constraint>. *S* shall not include a constraint descriptor whose constraint name is <constraint name>.

Access Rules

1) If a <domain definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <domain name>.

General Rules

1) A <domain definition> defines a domain.

2) A data type descriptor is created that describes the declared type of the domain being created.

3) A domain descriptor is created that describes the domain being created. The domain descriptor contains the name of the domain, the data type descriptor of the declared type, the value of the <default clause> if the <domain definition> immediately contains <default clause>, and a domain constraint descriptor for every immediately contained <domain constraint>.

4) A privilege descriptor is created that defines the USAGE privilege on this domain to the <authorization identifier> *A* of the schema or SQL-client module in which the <domain definition> appears. This privilege is grantable if and only if the applicable privileges for *A* include a grantable REFERENCES privilege for each column reference included in the domain descriptor and a grantable USAGE privilege for each <domain name>, <collation name>, <character set name>, and <transliteration name> contained in the <search condition> of any domain constraint descriptor included in the domain descriptor. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

5) A domain constraint descriptor is created that describes the domain constraint being defined. The domain constraint descriptor includes:

   a) The <constraint name> contained in the explicit or implicit <constraint name definition>.

   b) An indication of whether the constraint is deferrable or not deferrable.

   c) An indication of whether the initial constraint mode of the constraint is deferred or immediate.

   d) The <search condition> contained in the <domain definition> as the template <search condition>.

   e) The applicable <search condition>:

```
( 1=1 )
```

Conformance Rules

1) Without Feature F251, "Domain support", conforming SQL language shall not contain a <domain definition>.

2) Without Feature F692, “Extended collation support”, conforming SQL language shall not contain a <domain definition> that immediately contains a <collate clause>.

11.25 <alter domain statement>

Function

Change a domain and its definition.

Format

```
<alter domain statement> ::=
  ALTER DOMAIN <domain name> <alter domain action>

<alter domain action> ::=
    <set domain default clause>
  | <drop domain default clause>
  | <add domain constraint definition>
  | <drop domain constraint definition>
```

Syntax Rules

1) Let *D* be the domain identified by <domain name>. The schema identified by the explicit or implicit schema name of the <domain name> shall include the descriptor of *D*.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of <domain name>.

General Rules

1) The domain descriptor of *D* is modified as specified by <alter domain action>.

   NOTE 308 — The changed domain descriptor of *D* is applicable to every column that is dependent on *D*.

Conformance Rules

1) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain an <alter domain statement>.

11.26 <set domain default clause>

Function

Set the default value in a domain.

Format

```
<set domain default clause> ::=
  SET <default clause>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *D* be the domain identified by the <domain name> in the containing <alter domain statement>.

2) The default value specified by the <default clause> is placed in the domain descriptor of *D*.

Conformance Rules

1) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain a <set domain default clause>.

11.27 <drop domain default clause>

Function

Remove the default clause of a domain.

Format

```
<drop domain default clause> ::=
  DROP DEFAULT
```

Syntax Rules

1) Let *D* be the domain identified by the <domain name> in the containing <alter domain statement>.

2) The descriptor of *D* shall contain a default value.

Access Rules

None.

General Rules

1) Let *C* be the set of columns whose column descriptors contain the <domain name> that identifies *D*.

2) For every column belonging to *C*, if the column descriptor does not already contain a default value, then the default value from the domain descriptor of *D* is placed in that column descriptor.

3) The default value is removed from the domain descriptor of *D*.

Conformance Rules

1) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain a <drop domain default clause>.

11.28 <add domain constraint definition>

Function

Add a constraint to a domain.

Format

```
<add domain constraint definition> ::=
  ADD <domain constraint>
```

Syntax Rules

1) Let *D* be the domain identified by the <domain name> in the <alter domain statement>.

2) Let *D1* be some domain. *D1* is *in usage by* a domain constraint *DC* if and only if the <search condition> of *DC* generally contains the <domain name> either of *D1* or of some domain *D2* such that *D1* is in usage by some domain constraint of *D2*. No domain shall be in usage by any of its own constraints.

Access Rules

None.

General Rules

1) The constraint descriptor of the <domain constraint> is added to the domain descriptor of *D*.

2) If *DC* causes some column *CN* to be known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of the column descriptor of *CN* is changed to known not nullable.

NOTE 309 — The nullability characteristic of a column is defined in Subclause 4.13, “Columns, fields, and attributes”.

Conformance Rules

1) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain an <add domain constraint definition>.

11.29 <drop domain constraint definition>

Function

Destroy a constraint on a domain.

Format

```
<drop domain constraint definition> ::=
  DROP CONSTRAINT <constraint name>
```

Syntax Rules

1) Let *D* be the domain identified by the <domain name> *DN* in the containing <alter domain statement>.

2) Let *CD* be any column descriptor that includes *DN*, let *T* be the table described by the table descriptor that includes *CD*, and let *TN* be the <table name> of *T*.

3) Let *DC* be the descriptor of the constraint identified by <constraint name>.

4) *DC* shall be included in the domain descriptor of *D*.

Access Rules

None.

General Rules

1) The constraint descriptor *DC* is removed from the domain descriptor of *D*.

2) If *DC* causes some column *CN* to be known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of the column descriptor of *CN* is changed to possibly nullable.

   NOTE 310 — The nullability characteristic of a column is defined in Subclause 4.13, "Columns, fields, and attributes".

3) The descriptor of *DC* is destroyed.

Conformance Rules

1) Without Feature F711, "ALTER domain", conforming SQL language shall not contain a <drop domain constraint definition>.

2) Without Feature F491, "Constraint management", conforming SQL language shall not contain a <drop domain constraint definition>.

11.30 <drop domain statement>

This Subclause is modified by Subclause 9.13, "<drop domain statement>", in ISO/IEC 9075-4.

Function

Destroy a domain.

Format

```
<drop domain statement> ::=
  DROP DOMAIN <domain name> <drop behavior>
```

Syntax Rules

1) Let *D* be the domain identified by <domain name> and let *DN* be that <domain name>. The schema identified by the explicit or implicit schema name of *DN* shall include the descriptor of *D*.

2) If RESTRICT is specified, then *D* shall not be referenced in any of the following:

   a) A column descriptor.

   b) The <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) The SQL routine body of any routine descriptor.

3) Let *UA* be the <authorization identifier> that owns the schema identified by the <schema name> of the domain identified by *DN*.

Access Rules

1) The enabled authorization identifiers shall include *UA*.

General Rules

1) Let *C* be any column descriptor that includes *DN*, let *T* be the table described by the table descriptor that includes *C*, and let *TN* be the table name of *T*. *C* is modified as follows:

   a) *DN* is removed from *C*. A copy of the data type descriptor of *D* is included in *C*.

   b) If *C* does not include a <default clause> and the domain descriptor of *D* includes a <default clause>, then a copy of the <default clause> of *D* is included in *C*.

   c) For every domain constraint descriptor *DCD* included in the domain descriptor of *D*, for every domain constraint usage descriptor *DCU* included in *DCD*:

      i) Let *SC* be the applicable <search condition> included in *DCU*.

ii) Let *TCD* be a <table constraint definition> consisting of a <constraint name definition> whose <constraint name> is implementation-dependent, whose <constraint characteristics> are the <constraint characteristics> of the domain constraint descriptor, and whose <table constraint> is:

```
CHECK ( SC )
```

iii) If the applicable privileges for *UA* include all of the privileges necessary for *UA* to successfully execute the <alter table statement>

```
ALTER TABLE TN ADD TCD
```

then the following <alter table statement> is effectively executed with a current authorization identifier of *UA*:

```
ALTER TABLE TN ADD TCD
```

2) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE USAGE ON DOMAIN DN
FROM UA CASCADE
```

3) The descriptor of *D* is destroyed.

Conformance Rules

1) Without Feature F251, "Domain support", conforming SQL language shall not contain a <drop domain statement>.

11.31 <character set definition>

Function

Define a character set.

Format

```
<character set definition> ::=
  CREATE CHARACTER SET <character set name> [ AS ]
      <character set source> [ <collate clause> ]

<character set source> ::=
  GET <character set specification>
```

Syntax Rules

1) If a <character set definition> is contained in a <schema definition> and if the <character set name> immediately contained in the <character set definition> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <schema definition>.

2) The schema identified by the explicit or implicit schema name of the <character set name> shall not include a character set descriptor whose character set name is <character set name>.

3) The character set *CS* identified by the <character set specification> contained in <character set source> shall have associated with it a privilege descriptor that was effectively defined by the <grant statement>

   ```
   GRANT USAGE ON CHARACTER
   SET CSN TO PUBLIC
   ```

   where *CSN* is a <character set name> that identifies *CS*.

4) If <collate clause> is specified, then the <collation name> contained in <collate clause> shall identify a collation descriptor *CD* included in the schema identified by the explicit or implicit <schema name> contained in the <collation name>. The collation shall be applicable to the character repertoire of the character set identified by <character set source>. The list of applicable character set names included in *CD* shall include one that identifies *CS*.

5) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of <character set name>.

Access Rules

1) If a <character set definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include USAGE on the character set identified by the <character set specification>.

General Rules

1) A <character set definition> defines a character set.

2) A character set descriptor is created for the defined character set.

3) The descriptor created for the character set being defined is identical to the descriptor for the character set identified by <character set specification>, except that the included character set name is <character set name> and, if <collate clause> is specified, then the included name of the default collation is the <collation name> contained in <collate clause>.

4) A privilege descriptor is created that defines the USAGE privilege on this character set to be the <authorization identifier> of the <schema definition> or <SQL-client module definition> in which the <character set definition> appears. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM". This privilege is grantable.

Conformance Rules

1) Without Feature F451, "Character set definition", conforming SQL language shall not contain a <character set definition>.

11.32 <drop character set statement>

This Subclause is modified by Subclause 9.14, "<drop character set statement>", in ISO/IEC 9075-4.

Function

Destroy a character set.

Format

```
<drop character set statement> ::=
  DROP CHARACTER SET <character set name>
```

Syntax Rules

1) Let *C* be the character set identified by the <character set name> and let *CN* be the name of *C*.

2) The schema identified by the explicit or implicit schema name of *CN* shall include the descriptor of *C*.

3) The explicit or implicit <schema name> contained in *CN* shall not be equivalent to INFORMATION_SCHEMA.

4) *C* shall not be referenced in any of the following:

   a) The data type descriptor included in any column descriptor.

   b) The data type descriptor included in any domain descriptor.

   c) The data type descriptor generally included in any user-defined type descriptor.

   d) The data type descriptor included in any field descriptor.

   e) The <query expression> of any view descriptor.

   f) The <search condition> of any constraint descriptor.

   g) The collation descriptor of any collation.

   h) The transliteration descriptor of any transliteration.

   i) The SQL routine body, the <SQL parameter declaration>s, or the <returns data type> of any routine descriptor.

   j) The <SQL parameter declaration>s or <returns data type> of any method specification descriptor.

5) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <character set name>.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the character set identified by *C*. The enabled authorization identifiers shall include *A*.

General Rules

1) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE USAGE ON CHARACTER
SET CN FROM A
CASCADE
```

2) The descriptor of *C* is destroyed.

Conformance Rules

1) Without Feature F451, "Character set definition", conforming SQL language shall not contain a <drop character set statement>.

11.33 <collation definition>

Function

Define a collation.

Format

```
<collation definition> ::=
  CREATE COLLATION <collation name> FOR <character set specification>
      FROM <existing collation name> [ <pad characteristic> ]

<existing collation name> ::=
  <collation name>

<pad characteristic> ::=
    NO PAD
  | PAD SPACE
```

Syntax Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <collation name>.

2) If a <collation definition> is contained in a <schema definition> and if the <collation name> immediately contained in the <collation definition> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <schema definition>.

3) The schema identified by the explicit or implicit schema name of the <collation name> *CN* immediately contained in <collation definition> shall not include a collation descriptor whose collation name is *CN*.

4) The schema identified by the explicit or implicit schema name of the <collation name> *ECN* immediately contained in <existing collation name> shall include a collation descriptor whose collation name is *ECN*.

5) The collation identified by *ECN* shall be a collation whose descriptor includes a character repertoire name that is equivalent to that included in the descriptor of the character set identified by <character set specification>.

6) If <pad characteristic> is not specified, then the <pad characteristic> of the collation identified by *ECN* is implicit.

7) If NO PAD is specified, then the collation is said to have the NO PAD characteristic. If PAD SPACE is specified, then the collation is said to have the PAD SPACE characteristic.

Access Rules

1) If a <collation definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include USAGE on *ECN*.

General Rules

1) A <collation definition> defines a collation.

2) A privilege descriptor is created that defines the USAGE privilege on this collation for *A*. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

3) This privilege descriptor is grantable if and only if the USAGE privilege for *A* on the collation identified by *ECN* is grantable.

4) A collation descriptor is created for the defined collation.

5) The collation descriptor *CD* created is identical to the collation descriptor for *ECN*, except that the collation name included in *CD* is *CN* and, if <pad characteristic> is specified, then the pad characteristic included in *CD* is <pad characteristic>.

Conformance Rules

1) Without Feature F690, "Collation support", conforming SQL language shall not contain a <collation definition>.

11.34 <drop collation statement>

This Subclause is modified by Subclause 9.15, "<drop collation statement>", in ISO/IEC 9075-4.

Function

Destroy a collation.

Format

```
<drop collation statement> ::=
  DROP COLLATION <collation name> <drop behavior>
```

Syntax Rules

1) Let *C* be the collation identified by the <collation name> and let *CN* be the name of *C*.

2) The schema identified by the explicit or implicit schema name of *CN* shall include the descriptor of *C*.

3) The explicit or implicit <schema name> contained in *CN* shall not be equivalent to INFORMATION_SCHEMA.

4) If RESTRICT is specified, then *C* shall not be referenced in any of the following:

   a) Any character set descriptor.

   b) The triggered action of any trigger descriptor.

   c) The <query expression> of any view descriptor.

   d) The <search condition> of any constraint descriptor.

   e) The SQL routine body, the <SQL parameter declaration>s, or the <returns data type> of any routine descriptor.

   f) The <SQL parameter declaration>s or the <returns data type> of any method specification descriptor.

5) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the collation identified by *C*.

6) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <collation name>.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) For every character set descriptor *CSD* that includes *CN*, *CSD* is modified such that it does not include *CN*. If *CSD* does not include any collation name, then *CSD* is modified to indicate that it utilizes the default collation for its character repertoire.

2) For every data type descriptor *DD* that includes *CN*, *DD* is modified such that it includes the collation name of the character set collation of the character set of *DD*.

   NOTE 311 — This causes the column, domain, attribute, or field described by *DD* to revert to the default collation for its character set.

3) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE USAGE ON COLLATION
CN FROM A CASCADE
```

4) The descriptor of *C* is destroyed.

Conformance Rules

1) Without Feature F690, "Collation support", conforming SQL language shall not contain a <drop collation statement>.

11.35 <transliteration definition>

Function

Define a character transliteration.

Format

```
<transliteration definition> ::=
  CREATE TRANSLATION <transliteration name> FOR <source character set specification>
      TO <target character set specification> FROM <transliteration source>

<source character set specification> ::=
  <character set specification>

<target character set specification> ::=
  <character set specification>

<transliteration source> ::=
    <existing transliteration name>
  | <transliteration routine>

<existing transliteration name> ::=
  <transliteration name>

<transliteration routine> ::=
  <specific routine designator>
```

Syntax Rules

1) If a <transliteration definition> is contained in a <schema definition> and if the <transliteration name> immediately contained in the <transliteration definition> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <schema definition>.

2) The schema identified by the explicit or implicit schema name of the <transliteration name> *TN* immediately contained in <transliteration definition> shall not include a transliteration descriptor whose transliteration name is *TN*.

3) The schema identified by the explicit or implicit schema name of the <character set name> *SCSN* contained in the <character set specification> contained in <source character set specification> shall include a character set descriptor whose character set name is *SCSN*.

4) The schema identified by the explicit or implicit schema name of the <character set name> *TCSN* contained in the <character set specification> contained in <source character set specification> shall include a character set descriptor whose character set name is *TCSN*.

5) If <existing transliteration name> is specified, then:

   a) The schema identified by the explicit or implicit schema name of the <transliteration name> *TN* contained in <transliteration source> shall include a transliteration descriptor whose transliteration name is *TN*.

   b) The character set identified by *SCSN* shall have the same character repertoire and character encoding form as the source character set of the transliteration identified by *TN*.

c) The character set identified by *TCSN* shall have the same character repertoire and character encoding form as the target character set of the transliteration identified by *TN*.

6) If <transliteration routine> is specified, then:

a) The schema identified by the explicit or implicit schema name of the <specific routine designator> *SRD* contained in <transliteration routine> shall include a routine descriptor that identifies a routine having a <specific routine designator> *SRD*.

b) The routine identified by *SRD* shall be an SQL-invoked function that has one parameter whose data type is character string and whose character set is the character set specified by *SCSN*; the <returns type> of the routine shall be character string whose character set is the character set specified by *TCSN*.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of <transliteration name>. If a <transliteration definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) If <transliteration source> is specified, then the applicable privileges for *A* shall include USAGE on the transliteration identified by *TN*.

3) If <transliteration routine> is specified, then the applicable privileges for *A* shall include EXECUTE on the routine identified by *RN*.

General Rules

1) A <transliteration definition> defines a transliteration.

2) If <transliteration source> contains <existing transliteration name>, then let *SRDN* be the specific name included in the transliteration descriptor whose transliteration name is *TN*; otherwise, let *SRDN* be the specific name of the SQL-invoked routine identified by <transliteration routine>.

3) A transliteration descriptor is created that includes:

a) The name of the transliteration *TN*.

b) The name of the character set *SCSN* from which it translates.

c) The name of the character set *TCSN* to which it translates.

d) *SRDN*, the specific name of the SQL-invoked routine that performs the transliteration.

4) A privilege descriptor *PD* is created that defines the USAGE privilege on this transliteration to the <authorization identifier> of the <schema definition> or <SQL-client module definition> in which the <transliteration definition> appears. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

5) *PD* is grantable if and only if the USAGE privilege for the <authorization identifier> of the <schema definition> or <SQL-client module definition> in which the <transliteration definition> appears is also grantable on every character set identified by a <character set name> contained in the <transliteration definition>.

Conformance Rules

1) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <transliteration definition>.

11.36 <drop transliteration statement>

This Subclause is modified by Subclause 9.16, "<drop transliteration statement>", in ISO/IEC 9075-4.

Function

Destroy a character transliteration.

Format

```
<drop transliteration statement> ::=
  DROP TRANSLATION <transliteration name>
```

Syntax Rules

1) Let *T* be the transliteration identified by the <transliteration name> and let *TN* be the name of *T*.

2) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the transliteration identified by *T*.

3) The schema identified by the explicit or implicit schema name of *TN* shall include the descriptor of *T*.

4) *T* shall not be referenced in any of the following:

   a) The triggered action of any trigger descriptor.

   b) The <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) The collation descriptor of any collation.

   e) The transliteration descriptor of any translation.

   f) The SQL routine body of any routine descriptor.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

   ```
   REVOKE USAGE ON TRANSLATION TN FROM
   A CASCADE
   ```

2) The descriptor of *T* is destroyed.

Conformance Rules

1) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <drop transliteration statement>.

11.37 <assertion definition>

This Subclause is modified by Subclause 9.17, "<assertion definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.8, "<assertion definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 12.4, "<assertion definition>", in ISO/IEC 9075-14.

Function

Specify an integrity constraint.

Format

```
<assertion definition> ::=
  CREATE ASSERTION <constraint name>
      CHECK <left paren> <search condition> <right paren>
      [ <constraint characteristics> ]
```

Syntax Rules

1) If an <assertion definition> is contained in a <schema definition> and if the <constraint name> contains a <schema name>, then that <schema name> shall be equivalent to the explicit or implicit <schema name> of the containing <schema definition>.

2) The schema identified by the explicit or implicit schema name of the <constraint name> shall not include a constraint descriptor whose constraint name is <constraint name>.

3) If <constraint characteristics> is not specified, then INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

4) The <search condition> shall not contain a <host parameter name>, an <SQL parameter name>, an <embedded variable specification> or a <dynamic parameter specification>.

   NOTE 312 — <SQL parameter name> is excluded because of the scoping rules for <SQL parameter name>.

5) No <query expression> in the <search condition> shall reference a temporary table.

6) The <search condition> shall simply contain a <boolean value expression> that is retrospectively deterministic.

   NOTE 313 — "retrospectively deterministic" is defined in Subclause 6.34, "<boolean value expression>".

7) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the <assertion definition>. If an <assertion definition> is contained in an <SQL-client module definition>, then the enabled authorization identifier shall include *A*.

General Rules

1) An <assertion definition> defines an assertion. An assertion is a constraint.

2) Let *SC* be the <search condition> simply contained in the <assertion definition>.

3) An assertion descriptor is created that describes the assertion being defined. This descriptor includes:

   a) The <constraint name>.

   b) Whether the constraint is deferrable or not deferrable, as specified in <constraint characteristics>.

   c) The initial constraint mode, as specified in <constraint characteristics>.

   d) The applicable <search condition> *SC*.

4) If the character representation of *SC* cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — search condition too long for information schema*.

   NOTE 314 — The Information Schema is defined in ISO/IEC 9075-11.

5) If *SC* causes some column *CN* be to known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of *CN* is changed to known not nullable.

   NOTE 315 — The nullability characteristic of a column is defined in Subclause 4.13, “Columns, fields, and attributes”.

Conformance Rules

1) Without Feature F521, “Assertions”, conforming SQL language shall not contain an <assertion definition>.

2) Without Feature F672, “Retrospective check constraints”, conforming SQL language shall not contain an <assertion definition> that generally contains CURRENT_DATE, CURRENT_TIMESTAMP, or LOCALTIMESTAMP.

11.38 <drop assertion statement>

This Subclause is modified by Subclause 9.18, "<drop assertion statement>", in ISO/IEC 9075-4.

Function

Destroy an assertion.

Format

```
<drop assertion statement> ::=
  DROP ASSERTION <constraint name> [ <drop behavior> ]
```

Syntax Rules

1) Let *A* be the assertion identified by <constraint name> and let *AN* be the name of *A*.

2) The schema identified by the explicit or implicit schema name of *AN* shall include the descriptor of *A*.

3) If <drop behavior> is not specified, then RESTRICT is implicit.

4) If RESTRICT is specified or implied, then *AN* shall not be referenced in the SQL routine body of any routine descriptor.

5) If *QS* is a <query specification> that contains a column reference to a column *C* in its <select list> that is not contained in a <set function specification>, and if *G* is the set of columns defined by the <grouping column reference list> of *QS*, and if the assertion A is needed to conclude that *G* *C* is a known functional dependency in *QS*, then *QS* is said to be *dependent on A*.

6) If *V* is a view that contains a <query specification> that is dependent on *A*, then *V* is said to be *dependent on A*.

7) If *R* is an SQL routine whose <SQL routine body> contains a <query specification> that is dependent on *A*, then *R* is said to be *dependent on A*.

8) If *C* is a constraint or assertion whose <search condition> contains a <query specification> that is dependent on *A*, then *C* is said to be *dependent on A*.

9) If *T* is a trigger whose triggered action contains a <query specification> that is dependent on *A*, then *T* is said to be *dependent on A*.

10) If RESTRICT is specified or implicit, or <drop behavior> is not specified, then:

    a) No table constraint shall be dependent on *A*.

    b) No view shall be dependent on *A*.

    c) No SQL routine shall be dependent on *A*.

    d) No constraint or assertion shall be dependent on *A*.

    e) No trigger shall be dependent on *A*.

NOTE 316 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the <revoke statement> specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the <schema name> of the assertion identified by *AN*.

General Rules

1) Let *R* be any SQL-invoked routine whose routine descriptor contains the <constraint name> of *A* in the SQL routine body. Let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

2) Let *VN* be the table name of any view *V* that is dependent on *A*. The following <drop view statement> is effectively executed for every *V*:

```
DROP VIEW VN CASCADE
```

3) Let *SN* be the specific name of any SQL routine *SR* that is dependent on *A*, or that contains a reference to *A*. The following <drop routine statement> is effectively executed for every *SR*:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

4) Let *CN* be the constraint name of any constraint *C* that is dependent on *A*. Let *TN* be the name of the table constrained by *C*. The following <alter table statement> is effectively executed for every *C*:

```
ALTER TABLE TN DROP CONSTRAINT CN CASCADE
```

5) Let *AN2* be the assertion name of any assertion *A* that is dependent on *A*. The following <drop assertion statement> is effectively executed for every *A*:

```
DROP ASSERTION AN2 CASCADE
```

6) Let *TN* be the trigger name of any trigger *T* that is dependent on *A*. The following <drop trigger statement> is effectively executed for every *T*:

```
DROP TRIGGER TN
```

7) Let *SC* be the <search condition> included in the descriptor of *A*. If *SC* causes some column *CN* be to known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of *CN* is changed to possibly nullable.

NOTE 317 — The nullability characteristic of a column is defined in Subclause 4.13, “Columns, fields, and attributes”.

8) The descriptor of *A* is destroyed.

Conformance Rules

1) Without Feature F521, “Assertions”, conforming SQL language shall not contain a <drop assertion statement>.

11.39 <trigger definition>

This Subclause is modified by Subclause 9.19, "<trigger definition>", in ISO/IEC 9075-4.

Function

Define triggered SQL-statements.

Format

```
<trigger definition> ::=
  CREATE TRIGGER <trigger name> <trigger action time> <trigger event>
      ON <table name> [ REFERENCING <transition table or variable list> ]
      <triggered action>

<trigger action time> ::=
    BEFORE
  | AFTER

<trigger event> ::=
    INSERT
  | DELETE
  | UPDATE [ OF <trigger column list> ]

<trigger column list> ::=
  <column name list>

<triggered action> ::=
  [ FOR EACH { ROW | STATEMENT } ]
      [ WHEN <left paren> <search condition> <right paren> ]
      <triggered SQL statement>

<triggered SQL statement> ::=
    <SQL procedure statement>
  | BEGIN ATOMIC { <SQL procedure statement> <semicolon> }... END

<transition table or variable list> ::=
  <transition table or variable>...

<transition table or variable> ::=
    OLD [ ROW ] [ AS ] <old transition variable name>
  | NEW [ ROW ] [ AS ] <new transition variable name>
  | OLD TABLE [ AS ] <old transition table name>
  | NEW TABLE [ AS ] <new transition table name>

<old transition table name> ::=
  <transition table name>

<new transition table name> ::=
  <transition table name>

<transition table name> ::=
  <identifier>

<old transition variable name> ::=
```

```
  <correlation name>

<new transition variable name> ::=
  <correlation name>
```

Syntax Rules

1) Case:

   a) If a <trigger definition> is contained in a <schema definition> and if the <trigger name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

   b) If a <trigger definition> is contained in an <SQL-client module definition> and if the <trigger name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <SQL-client module definition>.

2) Let *TN* be the <table name> of a <trigger definition>. The table *T* identified by *TN* is the *subject table* of the <trigger definition>.

3) The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *T*.

4) The schema identified by the explicit or implicit <schema name> of a <trigger name> *TRN* shall not include a trigger descriptor whose trigger name is *TRN*.

5) *T* shall be a base table that is not a declared local temporary table.

6) If a <trigger column list> is specified, then:

   a) No <column name> shall appear more than once in the <trigger column list>.

   b) The <column name>s of the <trigger column list> shall identify columns of *T*.

7) If REFERENCING is specified, then:

   a) Let *OR*, *OT*, *NR*, and *NT* be the <old transition variable name>, <old transition table name>, <new transition variable name>, and <new transition table name>, respectively.

   b) OLD or OLD ROW, NEW or NEW ROW, OLD TABLE, and NEW TABLE shall be specified at most once each within the <transition table or variable list>.

   c) Case:

      i) If <trigger event> specifies INSERT, then neither OLD ROW nor OLD TABLE shall be specified.

      ii) If <trigger event> specifies DELETE, then neither NEW ROW nor NEW TABLE shall be specified.

   d) No two of *OR*, *OT*, *NR*, and *NT* shall be equivalent.

   e) Both *OR* and *NR* are range variables.

      NOTE 318 — “range variable” is defined in Subclause 4.14.6, “Operations involving tables”.

   f) The scope of *OR*, *OT*, *NR*, and *NT* is the <triggered action>, excluding any <SQL schema statement>s that are contained in the <triggered action>.

8) If neither FOR EACH ROW nor FOR EACH STATEMENT is specified, then FOR EACH STATEMENT is implicit.

9) If *OR* or *NR* is specified, then FOR EACH ROW shall be specified.

10) The <triggered action> shall not contain an <SQL parameter reference>, a <host parameter name>, a <dynamic parameter specification>, or an <embedded variable name>.

11) It is implementation-defined whether the <triggered SQL statement> shall not generally contain an <SQL transaction statement>, an <SQL connection statement>, an <SQL schema statement>, an <SQL dynamic statement>, or an <SQL session statement>.

12) If BEFORE is specified, then:

    a) It is implementation-defined whether the <triggered action> shall not generally contain an <SQL data change statement> or a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

    b) Neither OLD TABLE nor NEW TABLE shall be specified.

    c) The <triggered action> shall not contain a <field reference> that references a field in the new transition variable corresponding to a generated column of *T*.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <trigger name> of the <trigger definition>. If a <trigger definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* for *T* shall include TRIGGER.

3) If the <triggered action> *TA* of a <trigger definition> contains an <old transition table name> *OTTN*, an <old transition variable name> *OTVN*, a <new transition table name> *NTTN*, or a <new transition variable name> *NTVN*, and *TA* contains *OTTN*, *OTVN*, or *NTTN*, or if *TA* contains *NTVN*, then the applicable privileges for *TA* shall include SELECT.

General Rules

1) A <trigger definition> defines a trigger.

2) *OT* identifies the old transition table. *NT* identifies the new transition table. *OR* identifies the old transition variable. *NR* identifies the new transition variable.

   NOTE 319 — “old transition table”, “new transition table”, “old transition variable”, and “new transition variable” are defined in Subclause 4.38.1, “General description of triggers”.

3) The transition table identified by *OT* is the table associated with *OR*. The transition table identified by *NT* is the table associated with *NR*.

4) If the character representation of the <triggered SQL statement> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — statement too long for information schema*.

   NOTE 320 — The Information Schema is defined in ISO/IEC 9075-11.

5) A trigger descriptor is created for <trigger definition>s as follows:

a) The trigger name included in the trigger descriptor is <trigger name>.

b) The subject table included in the trigger descriptor is <table name>.

c) The trigger action time included in the trigger descriptor is <trigger action time>.

d) If FOR EACH STATEMENT is specified or implicit, then an indication that the trigger is a statement-level trigger is included in the trigger descriptor; otherwise, an indication that the trigger is a row-level trigger is included in the trigger descriptor.

e) The trigger event included in the trigger descriptor is <trigger event>.

f) Any <old transition variable name>, <new transition variable name>, <old transition table name>, or <new transition table name> specified in the <trigger definition> is included in the trigger descriptor as the old transition variable name, new transition variable name, old transition table name, or new transition table name, respectively.

g) The trigger action included in the trigger descriptor is the specified <triggered action>.

h) If a <trigger column list> *TCL* is specified, then *TCL* is the trigger column list included in the trigger descriptor; otherwise, that trigger column list is empty.

i) The *triggered action column set* included in the trigger descriptor is the set of all distinct, fully qualified names of columns contained in the <triggered action>.

j) The timestamp of creation included in the trigger descriptor is the timestamp of creation of the trigger.

Conformance Rules

1) Without Feature T211, “Basic trigger capability”, conforming SQL language shall not contain a <trigger definition>.

2) Without Feature T212, “Enhanced trigger capability”, in conforming SQL language, a <triggered action> shall contain FOR EACH ROW.

11.40 <drop trigger statement>

Function

Destroy a trigger.

Format

```
<drop trigger statement> ::=
  DROP TRIGGER <trigger name>
```

Syntax Rules

None.

Access Rules

1) Let *TR* be the trigger identified by the <trigger name>. The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the <schema name> of *TR*.

General Rules

1) The descriptor of *TR* is destroyed.

Conformance Rules

1) Without Feature T211, “Basic trigger capability”, conforming SQL language shall not contain a <drop trigger statement>.

11.41 <user-defined type definition>

This Subclause is modified by Subclause 11.9, "<user-defined type definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 9.4, "<user-defined type definition>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.5, "<user-defined type definition>", in ISO/IEC 9075-14.

Function

Define a user-defined type.

Format

```
<user-defined type definition> ::=
  CREATE TYPE <user-defined type body>

<user-defined type body> ::=
  <schema-resolved user-defined type name>
      [ <subtype clause> ]
      [ AS <representation> ]
      [ <user-defined type option list> ]
      [ <method specification list> ]

<user-defined type option list> ::=
  <user-defined type option> [ <user-defined type option>... ]

<user-defined type option> ::=
    <instantiable clause>
  | <finality>
  | <reference type specification>
  | <cast to ref>
  | <cast to type>
  | <cast to distinct>
  | <cast to source>

<subtype clause> ::=
  UNDER <supertype name>

<supertype name> ::=
  <path-resolved user-defined type name>

<representation> ::=
    <predefined type>
  |<member list>

<member list> ::=
  <left paren> <member> [ { <comma> <member> }... ] <right paren>

<member> ::=
  <attribute definition>

<instantiable clause> ::=
    INSTANTIABLE
  | NOT INSTANTIABLE

<finality> ::=
```

```
    FINAL
  | NOT FINAL

<reference type specification> ::=
    <user-defined representation>
  | <derived representation>
  | <system-generated representation>

<user-defined representation> ::=
  REF USING <predefined type>

<derived representation> ::=
  REF FROM <list of attributes>

<system-generated representation> ::=
  REF IS SYSTEM GENERATED

<cast to ref> ::=
  CAST <left paren> SOURCE AS REF <right paren> WITH <cast to ref identifier>

<cast to ref identifier> ::=
  <identifier>

<cast to type> ::=
  CAST <left paren> REF AS SOURCE <right paren> WITH <cast to type identifier>

<cast to type identifier> ::=
  <identifier>

<list of attributes> ::=
  <left paren> <attribute name> [ { <comma> <attribute name> }... ] <right paren>

<cast to distinct> ::=
  CAST <left paren> SOURCE AS DISTINCT <right paren>
      WITH <cast to distinct identifier>

<cast to distinct identifier> ::=
  <identifier>

<cast to source> ::=
  CAST <left paren> DISTINCT AS SOURCE <right paren>
      WITH <cast to source identifier>

<cast to source identifier> ::=
  <identifier>

<method specification list> ::=
  <method specification> [ { <comma> <method specification> }... ]

<method specification> ::=
    <original method specification>
  | <overriding method specification>

<original method specification> ::=
  <partial method specification> [ SELF AS RESULT ] [ SELF AS LOCATOR ]
      [ <method characteristics> ]

<overriding method specification> ::=
  OVERRIDING <partial method specification>
```

```
<partial method specification> ::=
  [ INSTANCE | STATIC | CONSTRUCTOR ]
      METHOD <method name> <SQL parameter declaration list>
      <returns clause>
      [ SPECIFIC <specific method name> ]

<specific method name> ::=
  [ <schema name> <period> ]<qualified identifier>

<method characteristics> ::=
  <method characteristic>...

<method characteristic> ::=
    <language clause>
  | <parameter style clause>
  | <deterministic characteristic>
  | <SQL-data access indication>
  | <null-call clause>
```

Syntax Rules

1) Let *UDTD* be the <user-defined type definition>, let *UDTB* be the <user-defined type body> immediately contained in *UDTD*, let *UDTN* be the <schema-resolved user-defined type name> immediately contained in *UDTB*, let *SN* be the specified or implicit <schema name> of *UDTN*, let *SS* be the SQL-schema identified by *SN*, and let *UDT* be the data type defined by *UDTD*.

2) If *UDTD* is contained in a <schema definition> and *UDTN* contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

3) *SS* shall not include a user-defined type descriptor or a domain descriptor whose name is equivalent to *UDTN*.

4) None of <instantiable clause>, <finality>, <reference type specification>, <cast to ref>, <cast to type>, <cast to distinct>, or <cast to source> shall be specified more than once.

5) Case:

   a) If <representation> specifies <predefined type>, then *UDTD* defines a *distinct type*.

   b) Otherwise, *UDTD* defines a *structured type*.

6) If <finality> specifies FINAL, then <instantiable clause> shall not specify NOT INSTANTIABLE.

7) If *UDTD* defines a distinct type, then:

   a) Let *PSDT* be the data type identified by <predefined type>.

      Case:

      i) If *PSDT* is an exact numeric type, then let *SDT* be an implementation-defined exact numeric type whose precision is equal to the precision of *PSDT* and whose scale is equal to the scale of *PSDT*.

      ii) If *PSDT* is an approximate numeric type, then let *SDT* be an implementation-defined approximate numeric type whose precision is equal to the precision of *PSDT*.

      iii) Otherwise, let *SDT* be *PSDT*.

b) <instantiable clause> shall not be specified.

c) If <finality> is not specified, then FINAL is implicit; otherwise, FINAL shall be specified.

d) <subtype clause> shall not be specified.

e) <reference type specification> shall not be specified.

f) Neither <cast to ref> nor <cast to type> shall be specified.

g) If <cast to distinct> is specified, then let *FNUDT* be <cast to distinct identifier>; otherwise, let *FNUDT* be the <qualified identifier> of *UDTN*.

h) If <cast to source> is specified, then let *FNSDT* be <cast to source identifier>; otherwise, the Syntax Rules of Subclause 9.7, "Type name determination", are applied to *SDT*, yielding an <identifier> *FNSDT*.

8) If *UDTD* specifies a structured type, then:

a) Neither <cast to distinct> nor <cast to source> shall be specified.

b) If <subtype clause> is specified, then <reference type specification> shall not be specified.

c) If neither <subtype clause> nor <reference type specification> are specified, then <system-generated representation> is implicit.

d) If <instantiable clause> is not specified, then INSTANTIABLE is implicit.

e) <finality> shall be specified.

f) The *originally-defined attributes* of *UDT* are those defined by <attribute definition>s contained in <member list>. No two originally-defined attributes of *UDT* shall have equivalent <attribute name>s.

g) For each <attribute definition> *ATD* contained in <member list>, let *AN* be the <attribute name> contained in *ATD* and let *DT* be the <data type> contained in *ATD*. The following <original method specification>s are implicit:

```
METHOD AN ( )
    RETURNS DT
    LANGUAGE SQL
    DETERMINISTIC
    CONTAINS SQL
  RETURNS NULL ON NULL INPUT
```

This is the *original method specification* of the observer function of attribute *AN*.

```
METHOD AN ( ATTR DT )
    RETURNS UDTN
    SELF AS RESULT
    LANGUAGE SQL
    DETERMINISTIC
    CONTAINS SQL
  CALLED ON NULL INPUT
```

This is the *original method specification* of the mutator function of attribute *AN*.

h) If <user-defined representation> is specified, then:

i) Let *BT* be <predefined type>. *BT* is the *representation type of the referencing type* of *UDT*.

ii) *BT* shall be exact numeric or a character string type that is not a large object string type.

iii) If <cast to ref> is specified, then let *FNREF* be <cast to ref identifier>; otherwise, let *FNREF* be the <qualified identifier> of *UDTN*.

iv) Case:

1) If <cast to type> is specified, then let *FNTYP* be <cast to type identifier>.

2) Otherwise, the Syntax Rules of Subclause 9.7, "Type name determination", are applied to *BT*, yielding an <identifier> *FNTYP*.

i) If <derived representation> is specified, then no two <attribute name>s in <list of attributes> shall be equivalent.

j) If <subtype clause> is specified, then:

i) <supertype name> shall not be equivalent to *UDTN*.

ii) The <supertype name> immediately contained in the <subtype clause> shall identify the descriptor of some structured type *SST*. *UDT* is a direct subtype of *SST*, and *SST* is a direct supertype of *UDT*.

iii) The descriptor of *SST* shall not include an indication that *SST* is final.

iv) The inherited attributes of *UDT* are the attributes described by the attribute descriptors included in the descriptor of *SST*.

v) If <member list> is specified, then no <attribute name> contained in <member list> shall have an attribute name that is equivalent to the attribute name of an inherited attribute.

vi) If the user-defined type descriptor of *SST* indicates that the referencing type of *SST* has a user-defined representation, then let *BT* be the data type described by the data type descriptor of the representation type of the referencing type of *SST* included in the user-defined type descriptor of *SST*.

1) If <cast to ref> is specified, then let *FNREF* be <cast to ref identifier>; otherwise, let *FNREF* be the <qualified identifier> of *UDTN*.

2) Case:

A) If <cast to type> is specified, then let *FNTYP* be <cast to type identifier>.

B) Otherwise, the Syntax Rules of Subclause 9.7, "Type name determination", are applied to *BT*, yielding an <identifier> *FNTYP*.

k) If <cast to ref> or <cast to type> is specified, then exactly one of the following shall be true:

i) <user-defined representation> is specified.

ii) <subtype clause> is specified and the user-defined type descriptor of the direct supertype of *UDT* indicates that the referencing type of the direct supertype of *UDT* has a user-defined representation.

9) If <method specification list> is specified, then:

a) Let M be the number of <method specification>s MS_i, 1 (one) $\leq i \leq M$, contained in <method specification list>. Let MN_i be the <method name> of MS_i.

b) For *i* ranging from 1 (one) to *M*:

i) If MS_i does not specify INSTANCE, CONSTRUCTOR, or STATIC, then INSTANCE is implicit.

ii) If MS_i specifies STATIC, then:

1) None of SELF AS RESULT, SELF AS LOCATOR, and OVERRIDING shall be specified.

2) MS_i specifies a *static method*.

iii) If MS_i specifies CONSTRUCTOR, then:

1) SELF AS RESULT shall be specified.

2) OVERRIDING shall not be specified.

3) MN_i shall be equivalent to the <qualified identifier> of *UDTN*.

4) The <returns data type> shall specify *UDTN*.

5) *UDTD* shall define a structured type.

6) MS_i specifies an *SQL-invoked constructor method*.

iv) Let RN_i be $SN.MN_i$.

v) If <specific method name> is not specified, then an implementation-dependent <specific method name> whose <schema name> is equivalent to *SN* is implicit.

vi) If <specific method name> contains a <schema name>, then that <schema name> shall be equivalent to *SN*. If <specific method name> does not contain a <schema name>, then the <schema name> of *SN* is implicit.

vii) The schema identified by the explicit or implicit <schema name> of the <specific method name> shall not include a routine descriptor whose specific name is equivalent to <specific method name> or a user-defined type descriptor that includes a method specification descriptor whose specific method name is equivalent to <specific method name>.

viii) Let PDL_i be the <SQL parameter declaration list> contained in MS_i.

1) No two <SQL parameter name>s contained in PDL_i shall be equivalent.

2) No <SQL parameter name> contained in PDL_i shall be equivalent to SELF.

ix) Let N_i be the number of <SQL parameter declaration>s contained in MS_i. For every <SQL parameter declaration> $PD_{i,j}$, 1 (one) $\le j \le N_i$:

1) $PD_{i,j}$ shall not contain <parameter mode>. A <parameter mode> of IN is implicit.

2) $PD_{i,j}$ shall not specify RESULT.

3) <parameter type> $PT_{i,j}$ immediately contained in $PD_{i,j}$ shall not specify ROW.

4) If $PT_{i,j}$ simply contains <locator indication>, then:

A) MS_i shall not specify or imply LANGUAGE SQL.

B) $PT_{i,j}$ shall specify either binary large object type, character large object type, array type, multiset type, or user-defined type.

x) If <returns data type> *RT* simply contains <locator indication>, then:

1) LANGUAGE SQL shall not be specified or implied.

2) *RT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

3) <result cast> shall not be specified.

xi) If SELF AS RESULT is specified, then the <returns data type> shall specify *UDTN*.

xii) For k ranging from (i+1) to M, at least one of the following conditions shall be false:

1) MN_i and the <method name> of MS_k are equivalent.

2) Both MS_i and MS_k either specify CONSTRUCTOR or neither specifies CONSTRUCTOR.

3) MS_k has N_i <SQL parameter declaration>s.

4) The data type of $PT_{i,j}$, 1 (one) $\leq j \leq N_i$, is compatible with $PT_{k,j}$.

xiii) The *unaugmented SQL parameter declaration list* of MS_i is the <SQL parameter declaration list> contained in MS_i.

xiv) If MS_i specifies <original method specification>, then:

1) The <method characteristics> of MS_i shall contain at most one <language clause>, at most one <parameter style clause>, at most one <deterministic characteristic>, at most one <SQL-data access indication>, and at most one <null-call clause>.

2) If <language clause> is not specified, then LANGUAGE SQL is implicit.

3) If <deterministic characteristic> is not specified, then NOT DETERMINISTIC is implicit.

4) <SQL-data access indication> shall be specified.

5) If <null-call clause> is not specified, then CALLED ON NULL INPUT is implicit.

6) Case:

A) If LANGUAGE SQL is specified or implied, then:

I) The <returns clause> shall not specify a <result cast>.

II) <SQL-data access indication> shall not specify NO SQL.

III) <parameter style clause> shall not be specified.

IV) Every <SQL parameter declaration> contained in <SQL parameter declaration list> shall contain an <SQL parameter name>.

B) Otherwise:

I) If <parameter style> is not specified, then PARAMETER STYLE SQL is implicit.

II) If a <result cast> is specified, then let *V* be some value of the <data type> specified in the <result cast> and let *RT* be the <returns data type>. The following shall be valid according to the Syntax Rules of Subclause 6.12, "<cast specification>":

```
CAST ( V AS RT )
```

III) If <result cast from type> *RCT* simply contains <locator indication>, then *RCT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

7) Let a *conflicting method specification CMS* be a method specification that is included in the descriptor of a proper supertype of *UDT*, such that the following are all true:

A) The method names of *CMS* and MN_i are equivalent.

B) *CMS* and MS_i have the same number of SQL parameters N_i.

C) Let $PCMS_j$, 1 (one) $\leq j \leq N_i$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of *CMS*. Let $PMS_{i,j}$, 1 (one) $\leq j \leq N_i$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of MS_i.

D) For *j* varying from 1 (one) to N_i, the declared type of $PCMS_j$ and the declared type of $PMS_{i,j}$ are compatible.

E) MS_i does not specify CONSTRUCTOR.

F) *CMS* and MS_i either both are not static methods or one of *CMS* and MS_i is a static method and the other is not a static method.

8) There shall be no conflicting method specification.

9) The *augmented SQL parameter declaration list* NPL_i of MS_i is defined as follows.

Case:

A) If MS_i specifies STATIC, then let NPL_i be:

```
( PD_i,1 , ..., PD_i,N_i )
```

B) If MS_i specifies SELF AS RESULT and SELF AS LOCATOR, then let NPL_i be:

```
( SELF UDTN RESULT AS LOCATOR, PD_i,1 , ..., PD_i,N_i )
```

C) If MS_i specifies SELF AS LOCATOR, then let NPL_i be:

```
( SELF UDTN AS LOCATOR, PD_i,1 , ..., PD_i,N_i )
```

D) If MS_i specifies SELF AS RESULT, then let NPL_i be:

```
( SELF UDTN RESULT, PD_i,1 , ..., PD_i,N_i )
```

E) Otherwise, let NPL_i be:

(SELF *UDTN*, $PD_{i,1}$, ..., PD_{i,N_i})

F) Let AN_i be the number of <SQL parameter declaration>s in NPL_i.

10) If MS_i does not specify STATIC or CONSTRUCTOR, then there shall be no SQL-invoked function *F* that satisfies all the following conditions:

A) The routine name of *F* and RN_i have equivalent <qualified identifier>s.

B) If *F* is not a static method, then *F* has AN_i SQL parameters; otherwise, *F* has (AN_i-1) SQL parameters.

C) The data type being defined is a proper subtype of

Case:

I) If *F* is not a static method, then the declared type of the first SQL parameter of *F*.

II) Otherwise, the user-defined type whose user-defined type descriptor includes the routine descriptor of *F*.

D) The declared type of the *i*-th SQL parameter in NPL_i, $2 \le i \le AN_i$ is compatible with

Case:

I) If *F* is not a static method, then the declared type of *i*-th SQL parameter of *F*.

II) Otherwise, the declared type of the (*i*-1)-th SQL parameter of *F*.

11) If MS_i specifies STATIC, then there shall be no SQL-invoked function *F* that is not a static method that satisfies all the following conditions:

A) The routine name of *F* and RN_i have equivalent <qualified identifier>s.

B) *F* has (AN_i+1) SQL parameters.

C) The data type being defined is a subtype of the declared type of the first SQL parameter of *F*.

D) The declared type of the *i*-th SQL parameter in *F*, $2 \le i \le (AN_i+1)$, is compatible with the declared type of the (*i*-1)-th SQL parameter of NPL_i.

xv) If MS_i specifies <overriding method specification>, then:

1) MS_i shall not specify STATIC or CONSTRUCTOR.

2) A <returns clause> contained in MS_i shall not specify a <result cast> or <locator indication>.

3) Let the candidate original method specification *COMS* be an original method specification whose descriptor is included in the descriptor of a proper supertype of the user-defined type being defined, such that the following are all true:

A) The <method name> of *COMS* and MN_i are equivalent.

B) *COMS* and MS_i have the same number of SQL parameters N_i.

C) Let $PCOMS_i$, 1 (one) ≤ i ≤ N_i, be the i-th SQL parameter in the unaugmented SQL parameter declaration list of *COMS*. Let $POVMS_i$, 1 (one) ≤ i ≤ N_i, be the i-th SQL parameter in the unaugmented SQL parameter declaration list of MS_i.

D) For i varying from 1 (one) to N_i, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with the declared type of $PCOMS_i$ and the declared type of $POVMS_i$.

E) The descriptor of *COMS* shall not include an indication that STATIC or CONSTRUCTOR was specified.

4) There shall exist exactly one *COMS*.

5) *COMS* shall not be the corresponding method specification of a mutator or observer function.

NOTE 321 — "Corresponding method specification" is defined in Subclause 11.50, "<SQL-invoked routine>".

6) For j ranging from 1 (one) to N_i, all of the following shall be true:

A) If $POVMS_j$ contains an <SQL parameter name> *PNM1*, then $PCOMS_j$ contains an <SQL parameter name> that is equivalent to *PNM1*.

B) If $PCOMS_j$ contains an <SQL parameter name> *PNM2*, then $POVMS_j$ contains an <SQL parameter name> that is equivalent to *PNM2*.

C) If $POVMS_j$ contains a <locator indication>, then $PCOMS_j$ contains a <locator indication>.

D) If $PCOMS_j$ contains a <locator indication>, then $POVMS_j$ contains a <locator indication>.

7) Let *ROVMS* be the <returns data type> of MS_i. Let *RCOMS* be the <returns data type> of *COMS*.

Case:

A) If *RCOMS* is a user-defined type, then:

I) Let a *candidate overriding method specification COVRMS* be a method specification that is included in the descriptor of a proper supertype of *UDT*, such that all of the following are true:

1) The <method name> of *COVRMS* and MN_i are equivalent.

2) *COVRMS* and MS_i have the same number of SQL parameters N_i.

3) Let $PCOVRMS_i$, 1 (one) ≤ i ≤ N_i, be the i-th SQL parameter in the unaugmented SQL parameter declaration list of *COVRMS*. Let $POVMS_i$, 1 (one) ≤ i ≤ N_i, be the i-th SQL parameter in the unaugmented SQL parameter declaration list of MS_i.

4) For i varying from 1 (one) to N_i, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with the declared type of $PCOVRMS_i$ and the declared type of $POVMS_i$.

II) Let *NOVMS* be the number of candidate overriding method specifications. For *i* varying from 1 (one) to *NOVMS*, *ROVMS* shall be a subtype of the <returns data type> of the *i*-th candidate overriding method specification.

B) Otherwise, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with *RCOMS* and *ROVMS*.

8) The augmented SQL parameter declaration list *ASPDL* of MS_i is formed from the augmented SQL parameter declaration list of *COMS* by replacing the <data type> of the first parameter (named SELF) with *UDTN*.

9) There shall be no SQL-invoked function *F* that satisfies all the following conditions:

A) The routine name of *F* and the RN_i have equivalent <qualified identifier>s.

B) *F* and *ASPDL* have the same number *N* of SQL parameters.

C) The data type being defined is a proper subtype of the declared type of the first SQL parameter of *F*.

D) The declared type of $POVMS_i$, 1 (one) $\leq i \leq N$, is compatible with the declared type of SQL parameter P_{i+1} of *F*.

E) *F* is not an SQL-invoked method.

Access Rules

1) Let *A* be the <authorization identifier> that owns *SS*. If a <user-defined type definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include UNDER on the <user-defined type name> specified in <subtype clause>.

General Rules

1) A user-defined type descriptor *UDTDS* that describes *UDT* is created. *UDTDS* includes:

a) The user-defined type name *UDTN*.

b) If *UDT* is a distinct type or INSTANTIABLE is specified or implicit, then an indication that *UDT* is instantiable; otherwise, an indication that *UDT* is not instantiable.

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

d) An indication of whether *UDT* is a distinct type or a structured type.

e) If *UDT* is a distinct type, then the data type descriptor of *SDT*.

f) If *UDT* is a structured type, then:

i) For each inherited attribute *IA* of *UDT*, the attribute descriptor of *IA* and an indication that *IA* is an inherited attribute.

ii) For each originally-defined attribute *ODA* of *UDT*, the attribute descriptor of *ODA* and an indication that *ODA* is an originally-defined attribute.

iii) The name of the direct supertype of *UDT*.

iv) A transform descriptor with an empty list of groups.

v) Case:

1) If <user-defined representation> is specified, then an indication that the referencing type of *UDT* has a user-defined representation, along with the data type descriptor of the representation type of the referencing type of *UDT*.

2) If <derived representation> is specified, then an indication that the referencing type of *UDT* has a derived representation, along with the attributes specified by <list of attributes>.

3) Otherwise, an indication that the referencing type of *UDT* has a system-defined representation.

vi) If <subtype clause> is specified, then let *SUDT* be the direct supertype of *UDT* and let *DSUDT* be the user-defined type descriptor of *SUDT*. Let *RUDT* be the referencing type of *UDT* and let *RSUDT* be the referencing type of *SUDT*.

Case:

1) If *DSUDT* indicates that *RSUDT* has a user-defined representation, then an indication that *RUDT* has a user-defined representation and the data type descriptor of the representation type of *RSUDT* included in *DSUDT*.

2) If *DSUDT* indicates that *RSUDT* has a derived representation, then an indication that *RUDT* has a derived representation and the list of attributes included in *DSUDT*.

3) If *DSUDT* indicates that *RSUDT* has a system-defined representation, then an indication that *RUDT* has a system-defined representation.

vii) The ordering form NONE.

viii) The ordering category STATE.

g) If <method specification list> is specified, then for every <original method specification> *ORMS* contained in <method specification list>, a method specification descriptor that includes:

i) An indication that the method specification is original.

ii) An indication of whether STATIC or CONSTRUCTOR is specified.

iii) The <method name> of *ORMS*.

iv) The <specific method name> of *ORMS*.

v) The <SQL parameter declaration list> contained in *ORMS* (augmented, if STATIC is not specified in *ORMS*, to include the implicit first parameter with parameter name SELF).

vi) The <language name> contained in the explicit or implicit <language clause>.

vii) The explicit or implicit <parameter style> if the <language name> is SQL.

viii) The <returns data type>.

ix) The <result cast from type>, if any.

x) An indication of whether the method is deterministic.

xi) An indication of whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

xii) An indication of whether the method should not be invoked if any argument is the null value.

h) If <method specification list> is specified, then for every <overriding method specification> *OVMS* contained in <method specification list>, let *DCMS* be the descriptor of the corresponding original method specification. The method specification descriptor of *OVMS* includes:

i) An indication that the method specification is overriding.

ii) The <method name> of *OVMS*.

iii) The <specific method name> of *OVMS*.

iv) The <SQL parameter declaration list> contained in *OVMS* (augmented to include the implicit first parameter with parameter name SELF).

v) The <language name> included in *DCMS*.

vi) The <parameter style> included in *DCMS* (if any).

vii) The <returns data type> of *OVMS*.

viii) The <result cast from type> included in *DCMS* (if any).

ix) The determinism indication included in *DCMS*.

x) The SQL-data access indication included in *DCMS*.

xi) The indication included in *DCMS*, whether the method should not be invoked if any argument is the null value.

2) If *UDTD* specifies a distinct type, then:

a) The degree of *UDT* is 0 (zero).

b) The following SQL-statements are executed without further Access Rule checking:

```
CREATE FUNCTION SN.FNUDT ( SDTP SDT )
    RETURNS UDTN
    LANGUAGE SQL
    DETERMINISTIC
  RETURN RV1
CREATE FUNCTION SN.FNSDT ( UDTP UDTN )
    RETURNS SDT
    LANGUAGE SQL
    DETERMINISTIC
  RETURN RV2
CREATE CAST ( UDTN AS SDT )
    WITH FUNCTION FNSDT ( UDTN )
    AS ASSIGNMENT
CREATE CAST (SDT AS UDTN)
    WITH FUNCTION SN.FNUDT ( SDT )
    AS ASSIGNMENT
CREATE TRANSFORM FOR UDTN
```

```
FNUDT ( FROM SQL WITH FUNCTION FNSDT ( UDTN ),
          TO SQL WITH FUNCTION SN.FNUDT(SDT) )
```

where: *SN* is the explicit or implicit <schema name> of *UDTN*; *RV1* is an implementation-dependent <value expression> such that for every invocation of *SN.FNUDT* with argument value *AV1*, *RV1* evaluates to the representation of *AV1* in the data type identified by *UDTN*; *RV2* is an implementation-dependent <value expression> such that for every invocation of *SN.FNSDT* with argument value *AV2*, *RV2* evaluates to the representation of *AV2* in the data type *SDT*, and *SDTP* and *UDTP* are <SQL parameter name>s arbitrarily chosen.

c) Case:

i) If *SDT* is not a large object type, then the following SQL-statement is executed without further Access Rule checking:

```
CREATE ORDERING FOR UDTN
ORDER FULL BY
MAP WITH FUNCTION FNSDT(UDTN)
FOR UDTN
```

ii) If *SDT* is a large object type, and the SQL implementation supports Feature T042, “Extended LOB data type support”, then the following SQL-statement is executed without further Access Rule checking:

```
CREATE ORDERING FOR UDTN
ORDER EQUALS ONLY BY
MAP WITH FUNCTION FNSDT(UDTN)
FOR UDTN
```

NOTE 322 — If *SDT* is a large object type, and the SQL implementation does not support Feature T042, “Extended LOB data type support”, then no ordering for *UDTN* is created.

3) If *UDTD* specifies a structured type, then:

a) The degree of *UDT* is the number of attributes of *UDT*, including inherited attributes. The ordinal position of an inherited attribute is its ordinal position in the direct supertype of *UDT*. The ordinal position of an attribute that is an originally-defined attribute is the ordinal position of its corresponding <attribute definition> in <member list> plus the number of inherited attributes.

b) If INSTANTIABLE is specified, then let *V* be a value of the most specific type *UDT* such that, for every attribute *ATT* of *UDT*, invocation of the corresponding observer function on *V* yields the default value for *ATT*. The following <SQL-invoked routine> is effectively executed:

```
CREATE FUNCTION UDTN () RETURNS UDTN
    RETURN V
```

This SQL-invoked function is the *constructor function* for *UDT*.

c) If <user-defined representation> is specified or if <subtype clause> is specified and the user-defined type descriptor of the direct supertype of *UDT* indicates that the referencing type of the direct supertype of *UDT* has a user-defined representation, then the following SQL-statements are executed without further Access Rule checking:

```
CREATE FUNCTION SN.FNREF ( BTP BT )
    RETURNS REF(UDTN)
    LANGUAGE SQL
    DETERMINISTIC
```

```
        STATIC DISPATCH
      RETURN RV1
    CREATE FUNCTION SN.FNTYP ( UDTNP REF(UDTN) )
        RETURNS BT
        LANGUAGE SQL
        DETERMINISTIC
        STATIC DISPATCH
      RETURN RV2
    CREATE CAST ( BT AS REF(UDTN) )
        WITH FUNCTION SN.FNREF(BT)
    CREATE CAST ( REF(UDTN) AS BT )
        WITH FUNCTION SN.FNTYP(REF(UDTN) )
```

where: *SN* is the explicit or implicit <schema name> of *UDTN*; *RV1* is an implementation-dependent <value expression> such that for every invocation of *SN.FNREF* with argument value *AV1*, *RV1* evaluates to the representation of *AV1* in the data type identified by REF(*UDTN*); *RV2* is an implementation-dependent <value expression> such that for every invocation of *SN.FNTYP* with argument value *AV2*, *RV2* evaluates to the representation of *AV2* in the data type *BT*; and *UDTNP* is an <SQL parameter name> arbitrarily chosen.

4) A privilege descriptor is created that defines the USAGE privilege on *UDT* to *A*. This privilege is grantable. The grantor for this privilege descriptor is set to the special grantor value "_SYSTEM".

5) If *UDTD* specifies a structured type, then a privilege descriptor is created that defines the UNDER privilege on *UDT* to *A*. The grantor for the privilege descriptor is set to the special grantor value "_SYSTEM". This privilege is grantable if and only if *A* holds the UNDER privilege on the direct supertype of *UDT* WITH GRANT OPTION.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <member list>.

2) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <instantiable clause> that contains NOT INSTANTIABLE.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <original method specification> that immediately contains SELF AS RESULT.

4) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <method characteristics> that contains a <parameter style> that contains GENERAL.

5) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <original method specification> that contains an <SQL-data access indication> that immediately contains NO SQL.

6) Without Feature T571, "Array-returning external SQL-invoked functions", conforming SQL language shall not contain a <method specification> that contains a <returns clause> that satisfies either of the following conditions:

   a) A <result cast from type> is specified that simply contains an <array type> and does not contain a <locator indication>.

   b) A <result cast from type> is not specified and <returns data type> simply contains an <array type> and does not contain a <locator indication>.

7) Without Feature T572, “Multiset-returning external SQL-invoked functions”, conforming SQL language shall not contain a <method specification> that contains a <returns clause> that satisfies either of the following conditions:

   a) A <result cast from type> is specified that simply contains a <multiset type> and does not contain a <locator indication>.

   b) A <result cast from type> is not specified and <returns data type> simply contains a <multiset type> and does not contain a <locator indication>.

8) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <reference type specification>.

9) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <partial method specification> that contains INSTANCE or STATIC.

10) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <method specification list>.

11) Without Feature S025, “Final structured types”, in conforming SQL language, a <user-defined type definition> that defines a structured type shall contain a <finality> that is NOT FINAL.

12) Without Feature S028, “Permutable UDT options list”, conforming SQL language shall not contain a <user-defined type option list> in which <instantiable clause>, if specified, <finality>, <reference type specification>, if specified, <cast to ref>, if specified, <cast to type>, if specified, <cast to distinct>, if specified, and <cast to source>, if specified, do not appear in that sequence.

11.42 <attribute definition>

This Subclause is modified by Subclause 9.5, "<attribute definition>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.6, "<attribute definition>", in ISO/IEC 9075-14.

Function

Define an attribute of a structured type.

Format

```
<attribute definition> ::=
  <attribute name> <data type>
      [ <attribute default> ]
      [ <collate clause> ]

<attribute default> ::=
  <default clause>
```

Syntax Rules

1) An <attribute definition> defines a certain component of some structured type. Let *UDT* be that structured type, let *UDTN* be its name, and let *SS* be the SQL-schema whose descriptor includes the descriptor of *UDT*.

2) Let *AN* be the <attribute name> contained in the <attribute definition>.

3) The declared type *DT* of the attribute is <data type>.

4) <collate clause> shall not be both specified in <data type> and immediately contained in <attribute definition>. If <collate clause> is immediately contained in <attribute definition>, then it is equivalent to specifying an equivalent <collate clause> in <data type>.

5) *DT* shall not be based on *UDT*.

   NOTE 323 — The notion of one data type being based on another data type is defined in Subclause 4.1, "Data types".

6) If *DT* is a <character string type> and does not contain a <character set specification>, then the default character set for *SS* is implicit.

Access Rules

None.

General Rules

1) A data type descriptor is created that describes *DT*.

2) Let *A* be the attribute defined by <attribute definition>.

3) An attribute descriptor is created that describes *A*. The attribute descriptor includes:

a) *AN*, the name of the attribute.

b) The data type descriptor of *DT*.

c) The ordinal position of the attribute in *UDT*.

d) The implicit or explicit <attribute default>.

e) The name *UDTN* of the user-defined type *UDT*.

4) An SQL-invoked method *OF* is created whose signature and result data type are as given in the descriptor of the original method specification of the observer function of *A*. Let *V* be a value in *UDT*. If *V* is the null value, then the invocation *V.AN*() of *OF* returns the result of:

```
CAST (NULL AS DT)
```

Otherwise, *V.AN*() returns the value of *A* in *V*.

NOTE 324 — The original method specification of the observer function of *A* is defined in the Syntax Rules of Subclause 11.41, "<user-defined type definition>".

NOTE 325 — The descriptor of *OF* is created under the General Rules of Subclause 11.50, "<SQL-invoked routine>".

5) An SQL-invoked method *MF* is created whose signature and result data type are as given in the descriptor of the original method specification of the mutator function of *A*. Let *V* be a value in *UDT* and let *AV* be a value in *DT*. If *V* is the null value, then the invocation *V.AN*(*AV*) of *MF* raises an exception condition: *data exception — null value substituted for mutator subject parameter*; otherwise, the invocation *V.AN*(*AV*) returns *V2* such that *V2.AN*() = *AV* and for every other observer function *ANX* of *UDT*, *V2.ANX*() = *V.ANX*().

NOTE 326 — The original method specification of the mutator function of *A* is defined in the Syntax Rules of Subclause 11.41, "<user-defined type definition>".

NOTE 327 — The descriptor of *MF* is created under the General Rules of Subclause 11.50, "<SQL-invoked routine>".

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain an <attribute definition>.

2) Without Feature F692, "Extended collation support", conforming SQL language shall not contain an <attribute definition> that immediately contains a <collate clause>.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <attribute default>.

4) Without Feature S026, "Self-referencing structured types", conforming SQL language shall not contain a <data type> simply contained in an <attribute definition> that is a <reference type> whose <referenced type> is equivalent to the <schema-resolved user-defined type name> simply contained in the <user-defined type definition> that contains <attribute definition>.

11.43 <alter type statement>

This Subclause is modified by Subclause 9.6, "<alter type statement>", in ISO/IEC 9075-13.

Function

Change the definition of a user-defined type.

Format

```
<alter type statement> ::=
  ALTER TYPE <schema-resolved user-defined type name> <alter type action>

<alter type action> ::=
    <add attribute definition>
  | <drop attribute definition>
  | <add original method specification>
  | <add overriding method specification>
  | <drop method specification>
```

Syntax Rules

1) Let *DN* be the <schema-resolved user-defined type name> and let *D* be the data type identified by *DN*.

2) The schema identified by the explicit or implicit schema name of the <schema-resolved user-defined type name> shall include the descriptor of *D*. Let *S* be that schema.

3) The scope of the <schema-resolved user-defined type name> is the entire <alter type statement>.

4) If <alter type action> contains <add attribute definition>, <drop attribute definition>, or <add overriding method specification>, then *D* shall be a structured type.

5) Let *A* be the <authorization identifier> that owns the schema *S*.

Access Rules

1) If an <alter type statement> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include UNDER on each proper supertype of *D*.

General Rules

1) The user-defined type descriptor of *D* is modified as specified by <alter type action>.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <alter type statement>.

11.44 <add attribute definition>

Function

Add an attribute to a user-defined type.

Format

```
<add attribute definition> ::=
  ADD ATTRIBUTE <attribute definition>
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> immediately contained in the containing <alter type statement>. Let *SPD* be any supertype of *D*. Let *SBD* be any subtype of *D*.

2) Let *RD* be the reference type whose referenced type is *D*. Let *SPRD* be any supertype of *RD*. Let *SBRD* be any subtype of *RD*. Let *AD* be any collection type whose element type is *D*. Let *SPAD* be any collection type whose element type is *SPD* or *SPRD*. Let *SBAD* be any collection type whose element type is *SBD* or *SBRD*.

3) The declared type of a column of a base table shall not be *SPRD*, *SBRD*, *SPAD*, or *SBAD*.

4) The declared type of a column of a base table shall not be based on *D*.

   NOTE 328 — The notion of one data type being based on another data type is defined in Subclause 4.1, "Data types".

5) *SBD* shall not be the structured type of a referenceable table.

6) Let *M* be the mutator function resulting from the <attribute definition>, had that <attribute definition> been simply contained in the <user-defined type definition> for *D*. There shall be no SQL-invoked routine *F* that satisfies all of the following conditions:

   a) The routine name included in the descriptor of *F* and the <schema qualified routine name> of *M* have equivalent <qualified identifier>s.

   b) *F* has two SQL parameters.

   c) The declared type of the first SQL parameter of *F* is a subtype or supertype of *D*.

   d) The declared type of the second SQL parameter of *F* is a compatible with the second SQL parameter of *M*.

7) Let *O* be the observer function resulting from the <attribute definition>, had that <attribute definition> been simply contained in the <user-defined type definition> for *D*. There shall be no SQL-invoked routine *F* that satisfies all of the following conditions:

   a) The <schema qualified routine name> of *O* and the routine name included in the descriptor of *F* have equivalent <qualified identifier>s.

   b) *F* has 1 (one) SQL parameter.

   c) The declared type of the first SQL parameter of *F* is a subtype or supertype of *D*.

Access Rules

None.

General Rules

1) The attribute defined by the <attribute definition> is added to *D*.

2) In all other respects, the specification of an <attribute definition> in an <alter type statement> has the same effect as specification of the <attribute definition> simply contained in the <user-defined type definition> for *D* would have had. In particular, the degree of *D* is increased by 1 (one) and the ordinal position of that attribute is equal to the new degree of *D* as specified in the General Rules of Subclause 11.42, "<attribute definition>".

3) Let *A* be the attribute defined by <attribute definition>. Let *CPA* be a copy of the descriptor of *A*, modified to include an indication that the attribute is an inherited attribute.

4) For each proper subtype *PSBD* of *D*:

   a) Let *DPSBD* be the descriptor of *PSBD*, let *N* be the number of attribute descriptors included in *DPSBD*, and let DA_i, 1 (one) $\leq i \leq N$, be the attribute descriptors included in *DPSBD*.

   b) For every *i* between 1 (one) and *N*, if DA_i is the descriptor of an originally-defined attribute, then increase the ordinal position included in DA_i by 1 (one).

   c) Include *CPA* in *DPSBD*.

Conformance Rules

None.

11.45 <drop attribute definition>

Function

Destroy an attribute of a user-defined type.

Format

```
<drop attribute definition> ::=
  DROP ATTRIBUTE <attribute name> RESTRICT
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> immediately contained in the containing <alter type statement>.

2) Let *A* be the attribute identified by the <attribute name> *AN*.

3) *A* shall be an attribute of *D* that is not an inherited attribute, and *A* shall not be the only attribute of *D*.

4) Let *SPD* be any supertype of *D*. Let *SBD* be any subtype of *D*. Let *RD* be the reference type whose referenced type is *D*. Let *SPRD* be any supertype of *RD*. Let *SBRD* be any subtype of *RD*. Let *AD* be any collection type whose element type is *D*. Let *SPAD* be any collection type whose element type is *SPD* or *SPRD*. Let *SBAD* be any collection type whose element type is *SBD* or *SBRD*.

5) The declared type of any column of any base table shall not be *SPRD*, *SBRD*, *SPAD*, or *SBAD*.

6) The declared type of any column of any base table shall not be based on *D*.

   NOTE 329 — The notion of one data type being based on another data type is defined in Subclause 4.1, "Data types".

7) *SBD* shall not be the structured type of a referenceable table.

8) Let *R1* be the mutator function and let *R2* be the observer function of *A*.

   a) *R1* and *R2* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The <query expression> of any view descriptor.

      iii) The <search condition> of any constraint descriptor.

      iv) The trigger action of any trigger descriptor.

   b) The specific names of *R1* and *R2* shall not be included in any user-defined cast descriptor.

   c) *R1* and *R2* shall not be the ordering function in the descriptor of any user-defined type.

Access Rules

None.

General Rules

1) The descriptor of *A* is removed from the descriptor of every *SBD*.

2) The descriptor of *A* is destroyed.

3) The descriptors of the mutator and observer functions of *A* are destroyed.

4) The degree of every *SBD* is reduced by 1 (one). The ordinal position of all attributes having an ordinal position greater than the ordinal position of *A* in *SBD* is reduced by 1 (one).

Conformance Rules

None.

11.46 <add original method specification>

Function

Add an original method specification to a user-defined type.

Format

```
<add original method specification> ::=
  ADD <original method specification>
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> *DN* immediately contained in the containing <alter type statement>. Let *SN* be the specified or implied <schema name> of *DN*. Let *SPD* be any supertype of *D*, if any. Let *SBD* be any subtype of *D*, if any.

2) Let *ORMS* and *PORMS* be the <original method specification> and its immediately contained <partial method specification>, respectively.

3) Let *MN*, *MPDL* and *MCH* be the <method name>, the <SQL parameter declaration list> and the <method characteristics>, respectively, that are simply contained in *ORMS*. *MPDL* is called the *unaugmented SQL parameter declaration list* of *ORMS*.

4) If *PORMS* does not specify INSTANCE, CONSTRUCTOR, or STATIC, then INSTANCE is implicit.

5) If *PORMS* specifies CONSTRUCTOR, then:

   a) SELF AS RESULT shall be specified.

   b) *MN* shall be equivalent to the <qualified identifier> of *DN*.

   c) The <returns data type> shall specify *DN*.

   d) *D* shall be a structured type.

   e) *PORMS* specifies an *SQL-invoked constructor method*.

6) If *PORMS* specifies STATIC, then:

   a) Neither SELF AS RESULT nor SELF AS LOCATOR shall be specified.

   b) *PORMS* specifies a static method.

7) Let *RN* be *SN*.*MN*.

8) Case:

   a) If *PORMS* does not specify <specific method name>, then an implementation-dependent <specific method name> is implicit whose <schema name> is equivalent to *SN*.

   b) Otherwise,

      Case:

i) If <specific method name> contains a <schema name>, then that <schema name> shall be equivalent to *SN*.

ii) Otherwise, the <schema name> *SN* is implicit.

The schema identified by the explicit or implicit <schema name> of the <specific method name> shall not include a routine descriptor whose specific name is equivalent to <specific method name> or a user-defined type descriptor that includes a method specification descriptor whose specific method name is equivalent to <specific method name>.

9) *MCH* shall contain at most one <language clause>, at most one <parameter style clause>, at most one <deterministic characteristic>, at most one <SQL-data access indication>, and at most one <null-call clause>.

a) If <language clause> is not specified in *MCH*, then LANGUAGE SQL is implicit.

b) Case:

i) If LANGUAGE SQL is specified or implied, then:

1) <parameter style clause> shall not be specified.

2) <SQL-data access indication> shall not specify NO SQL.

3) Every <SQL parameter declaration> contained in <SQL parameter declaration list> shall contain an <SQL parameter name>.

4) The <returns clause> shall not specify a <result cast>.

ii) Otherwise:

1) If <parameter style clause> is not specified, then PARAMETER STYLE SQL is implicit.

2) If a <result cast> is specified, then let *V* be some value of the <data type> specified in the <result cast> and let *RT* be the <returns data type>. The following shall be valid according to the Syntax Rules of Subclause 6.12, "<cast specification>":

```
CAST ( V AS RT )
```

3) If <result cast from type> *RCT* simply contains <locator indication>, then *RCT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

c) If <deterministic characteristic> is not specified in *MCH*, then NOT DETERMINISTIC is implicit.

d) If <SQL-data access indication> is not specified, then CONTAINS SQL is implicit.

e) If <null-call clause> is not specified in *MCH*, then CALLED ON NULL INPUT is implicit.

10) No two <SQL parameter name>s contained in *MPDL* shall be equivalent.

11) No <SQL parameter name> contained in *MPDL* shall be equivalent to SELF.

12) Let *N* be the number of <SQL parameter declaration>s contained in *MPDL*. For every <SQL parameter declaration> PD_j, 1 (one) $\leq j \leq N$:

a) PD_j shall not contain <parameter mode>. A <parameter mode> of IN is implicit.

b) PD_j shall not specify RESULT.

c) <parameter type> PT_j immediately contained in PD_j shall not specify ROW.

d) If PT_j simply contains <locator indication>, then:

i) *MCH* shall not specify LANGUAGE SQL, nor shall LANGUAGE SQL be implied.

ii) PT_j shall specify either binary large object type, character large object type, array type, multiset type, or user-defined type.

13) If <returns data type> *RT* simply contains <locator indication>, then:

a) *MCH* shall not be specify LANGUAGE SQL, nor shall LANGUAGE SQL be implied.

b) *RT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

c) <result cast> shall not be specified.

14) If SELF AS RESULT is specified, then the <returns data type> shall specify *DN*.

15) Case:

a) If *ORMS* specifies CONSTRUCTOR, then let a *conflicting method specification CMS* be a method specification whose descriptor is included in the descriptor of *D*, such that the following are all true:

i) *MPDL* and the unaugmented SQL parameter list of *CMS* have the same number *N* of SQL parameters.

ii) Let $PCMS_j$, 1 (one) $\leq j \leq N$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of *CMS*. Let PMS_j, 1 (one) $\leq j \leq N$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list *MPDL*.

iii) For *j* varying from 1 (one) to *N*, the declared type of $PCMS_j$ and the declared type of PMS_j are compatible.

iv) *CMS* is an SQL-invoked constructor method.

b) Otherwise, let a *conflicting method specification CMS* be a method specification whose descriptor is included in the descriptor of some *SPD* or *SBD*, such that the following are all true:

i) *MN* and the method name included in the descriptor of *CMS* are equivalent.

ii) *MPDL* and the unaugmented SQL parameter list of *CMS* have the same number *N* of SQL parameters.

iii) Let $PCMS_j$, 1 (one) $\leq j \leq N$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of *CMS*. Let PMS_j, 1 (one) $\leq j \leq N$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list *MPDL*.

iv) For *j* varying from 1 (one) to *N*, the declared type of $PCMS_j$ and the declared type of PMS_j are compatible.

v) *CMS* and *ORMS* either both are not instance methods or one of *CMS* and *ORMS* is a static method and the other is an instance method.

16) There shall be no conflicting method specification.

17) Let MP_i, 1 (one) ≤ i ≤ N, be the i-th <SQL parameter declaration> contained in *MPDL*. The augmented SQL parameter declaration list *NPL* of *ORMS* is defined as follows.

Case:

a) If *PORMS* specifies STATIC, then let *NPL* be:

(MP_1, ..., MP_N)

b) If *ORMS* specifies SELF AS RESULT and SELF AS LOCATOR, then let *NPL* be:

(SELF *DN* RESULT AS LOCATOR,
MP_1, ...,
MP_N)

c) If *ORMS* specifies SELF AS LOCATOR , then let *NPL* be:

(SELF *DN* AS LOCATOR,
MP_1, ...,
MP_N)

d) If *ORMS* specifies SELF AS RESULT, then let *NPL* be:

(SELF *DN* RESULT, MP_1 , ...,
MP_N)

e) Otherwise, let *NPL* be:

(SELF *DN*, MP_1, ...,
MP_N)

Let *AN* be the number of <SQL parameter declaration>s in *NPL*.

18) If *PORMS* does not specify STATIC or CONSTRUCTOR, then there shall be no SQL-invoked function *F* that satisfies all the following conditions:

a) *F* is not an SQL-invoked method.

b) The <routine name> of *F* and *RN* have equivalent <qualified identifier>s.

c) *F* has *AN* SQL parameters.

d) *D* is a subtype or supertype of the declared type of the first SQL parameter of *F*.

e) The declared type of the i-th SQL parameter in *NPL*, 2 ≤ i ≤ *AN* is compatible with the declared type of i-th SQL parameter of *F*.

19) If *PORMS* specifies STATIC, then there shall be no SQL-invoked function *F* that is not a static method that satisfies all the following conditions:

a) The <routine name> of *F* and *RN* have equivalent <qualified identifier>s.

b) *F* has (*AN*+1) SQL parameters.

c) *D* is a subtype or supertype of the declared type of the first SQL parameter of *F*.

d) The declared type of the *i*-th SQL parameter of *F*, $2 \le i \le (AN+1)$, is compatible with the declared type of the (*i*-1)-th SQL parameter of *NPL*.

Access Rules

None.

General Rules

1) Let *STDS* be the descriptor of *D*. A method specification descriptor *DOMS* is created for *ORMS*. *DOMS* includes:

   a) An indication that the method specification is original.

   b) An indication of whether STATIC or CONSTRUCTOR is specified.

   c) The <method name> *MN*.

   d) The <specific method name> contained in *PORMS*.

   e) The augmented SQL parameter declaration list *NPL*.

   f) For every parameter descriptor of a parameter of *NPL*, a locator indication (if specified).

   g) The <returns data type> contained in *PORMS*.

   h) The <result cast from type> contained in *PORMS* (if any).

   i) The locator indication, if a <locator indication> is contained in the <returns clause> of *PORMS* (if any).

   j) The <language name> explicitly or implicitly contained in *MCH*.

   k) The explicit or implicit <parameter style> contained in *MCH*, if the <language name> is not SQL.

   l) The determinism indication contained in *MCH*.

   m) An indication of whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

   n) An indication of whether the method should not be invoked if any argument is the null value.

2) *DOMS* is added to *STDS*.

3) Let *N* be the number of table descriptors that include the user-defined type name of a subtype of *D*.

   For *i* varying from 1 (one) to *N*:

   a) Let TN_i be the <table name> included in the *i*-th such table descriptor.

b) For every table privilege descriptor that specifies TN_i and a privilege of SELECT, a new table/method privilege descriptor is created that specifies TN_i, the same action, grantor, and grantee, and the same grantability, and the <specific method name> contained in *ORMS*.

Conformance Rules

None.

11.47 <add overriding method specification>

Function

Add an overriding method specification to a user-defined type.

Format

```
<add overriding method specification> ::=
  ADD <overriding method specification>
```

Syntax Rules

1) Let *OVMS* be the <overriding method specification> immediately contained in <add overriding method specification>. Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> *DN* immediately contained in the <alter type statement> containing *OVMS*. Let *SN* be the specified or implied <schema name> of *DN*. Let *SPD* be any supertype of *D*, if any. Let *SBD* be any subtype of *D*, if any.

2) Let *POVMS* be the <partial method specification> immediately contained in *OVMS*. *POVMS* shall not specify STATIC or CONSTRUCTOR.

3) Let *MN*, *RTC* and *MPDL* be <routine name>, the <returns clause> and the <SQL parameter declaration list> immediately contained in *POVMS*.

4) *MN* shall not be equivalent to the <qualified identifier> of the user-defined type name of any *SPD* or *SBD* other than *D*.

5) Let *RN* be *SN*.*MN*.

6) Case:

   a) If *POVMS* does not specify <specific method name>, then an implementation-dependent <specific method name> is implicit whose <schema name> is equivalent to *SN*.

   b) Otherwise,

      Case:

      i) If <specific method name> contains a <schema name>, then that <schema name> shall be equivalent to *SN*.

      ii) Otherwise, the <schema name> *SN* is implicit.

7) The schema identified by the explicit or implicit <schema name> of the <specific method name> shall not include a routine descriptor whose specific name is equivalent to <specific method name> or a user-defined type descriptor that includes a method specification descriptor whose specific method name is equivalent to <specific method name>.

8) *RTC* shall not specify a <result cast> or <locator indication>.

9) Let the *candidate original method specification COMS* be an original method specification that is included in the descriptor of a proper supertype of the user-defined type of *D*, such that the following are all true:

a) *MN* and the <method name> of *COMS* are equivalent.

b) Let *N* be the number of elements of the augmented SQL parameter declaration list *UPCOMS* generally included in the descriptor of *COMS*. *MPDL* contains (*N*-1) SQL parameter declarations.

c) For *i* varying from 2 to *N*, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with the data types of the SQL parameters $PCOMS_i$ of *UPCOMS* and the data types of the SQL parameters $POVMS_{i-1}$ of *MPDL*, respectively.

d) The descriptor of *COMS* shall not include an indication that STATIC or CONSTRUCTOR was specified.

10) There shall exist exactly one such *COMS*.

11) *COMS* shall not be the corresponding method specification of a mutator or observer function.

NOTE 330 — "Corresponding method specification" is defined in Subclause 11.50, "<SQL-invoked routine>".

12) For *i* varying from 2 to *N*:

a) If $POVMS_{i-1}$ contains an <SQL parameter name> *PNM1*, then the descriptor of the *i*-th parameter of the augmented <SQL parameter declaration list> of *UPCOMS* shall include a parameter name that is equivalent to *PNM1*.

b) If the descriptor of the *i*-th parameter of the augmented <SQL parameter declaration list> of *UPCOMS* includes a parameter name *PNM2*, then $POVMS_{i-1}$ shall contain an <SQL parameter name> that is equivalent to *PNM2*.

c) $POVMS_{i-1}$ shall not contain <parameter mode>. A <parameter mode> IN is implicit.

d) $POVMS_{i-1}$ shall not specify RESULT.

e) If the <parameter type> PT_{i-1} immediately contained in $POVMS_{i-1}$ contains a <locator indication>, then the descriptor of the *i*-th parameter of the augmented <SQL parameter declaration list> of *UPCOMS* shall include a <locator indication>.

f) If the the descriptor of the *i*-th parameter of the augmented <SQL parameter declaration list> of *UPCOMS* includes a <locator indication>, then the <parameter type> PT_{i-1} immediately contained in $POVMS_{i-1}$ shall contain a <locator indication>.

13) Let *ROVMS* be the <returns data type> of *RTC*. Let *RCOMS* be the <returns data type> of *COMS*.

Case:

a) If *RCOMS* is a user-defined type, then:

i) Let a *candidate overriding method specification COVRMS* be a method specification that is included in the descriptor of a proper supertype or a proper subtype of *UDT*, such that all of the following are true:

1) The <method name> of *COVRMS* and *MN* are equivalent.

2) *COVRMS* and *OVMS* have the same number of SQL parameters N_i.

3) Let $PCOVRMS_i$, 1 (one) $\le i \le N_i$, be the *i*-th SQL parameter in the unaugmented SQL parameter declaration list of *COVRMS*. Let $POVMS_i$, 1 (one) $\le i \le N_i$, be the *i*-th SQL parameter in the unaugmented SQL parameter declaration list of *OVMS*.

4) For i varying from 1 (one) to N_i, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with the declared type of $PCOVRMS_i$ and the declared type of $POVMS_i$.

ii) Let *NOVMS* be the number of candidate overriding method specifications. For i varying from 1 (one) to *NOVMS*, let $COVRMS_i$ be the i-th candidate overriding method specification.

Case:

1) If $COVRMS_i$ is included in the descriptor of a proper supertype of *D*, then *ROVMS* shall be a subtype of the <returns data type> of $COVRMS_i$.

2) Otherwise, *ROVMS* shall be a supertype of the <returns data type> of $COVRMS_i$.

b) Otherwise, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with *RCOMS* and *ROVMS* as the data types.

14) Let a *conflicting overriding method specification COVMS* be an overriding method specification that is included in the descriptor of *D*, such that all of the following are true:

a) *MN* and the method name of *COVMS* are equivalent.

b) The augmented SQL parameter declaration list of *COVMS* contains *N* elements.

c) For i varying from 2 to *N*, the data types of the SQL parameter $POVMS_{i-1}$ and the SQL parameter $PCOVMS_{i-1}$ of *COVMS* are compatible.

15) There shall be no conflicting overriding method specification.

16) The augmented SQL parameter declaration list *ASPDL* of *OVMS* is formed from the augmented SQL parameter declaration list of *COMS* by replacing the <data type> of the first parameter (named SELF) with the <schema-resolved user-defined type name> *DN*.

17) There shall be no SQL-invoked function *F* that satisfies all the following conditions:

a) *F* is not an SQL-invoked method.

b) The <routine name> of *F* and the <routine name> *MS* have equivalent <qualified identifier>s.

c) Let *NPF* be the number of SQL parameters in *ASPDL*. *F* has *NPF* SQL parameters.

d) *D* is a subtype or supertype of the declared type of the first SQL parameter of *F*.

e) The declared type of the i-th SQL parameter in *ASPDL*, $2 \le i \le NPF$ is compatible with the declared type of i-th SQL parameter of *F*.

18) If the descriptor of *D* includes any method specification descriptor, then:

a) Let *M* be the number of method specification descriptors MSD_i, 1 (one) $\le i \le M$, included in the descriptor of *D*.

b) For i ranging from 1 (one) to *M*:

i) Let N_i be the number of <SQL parameter declaration>s contained in the augmented SQL parameter declaration list included in MSD_i. Let $PT_{i,j}$, 1 (one) $\le j \le N_i$, be the j-th <parameter type> contained in MSD_i.

ii) At least one of the following conditions shall be false:

1) The <routine name> included in MSD_i is equivalent to *MN*.

2) *ASPDL* has N_i <SQL parameter declaration>s.

3) The data type of $PT_{i,j}$, 1 (one) $\leq j \leq N_i$, is compatible with the data type of the *j*-th <SQL parameter declaration> of *MPDL*.

4) MSD_i does not include an indication that CONSTRUCTOR was specified.

Access Rules

None.

General Rules

1) Let *STDS* be the descriptor of D, and *DCMS* the descriptor of the corresponding original method specification *COMS*. A method specification descriptor *DOMS* is created for *OVMS*. *DOMS* includes:

   a) An indication that the method specification is overriding.

   b) The <method name> *MN*.

   c) The <specific method name> contained in *POVMS*.

   d) The augmented SQL parameter declaration list *APDL*.

   e) For every parameter descriptor of a parameter of *APDL*, the locator indication of the descriptor of the corresponding parameter included in *DCMS* (if any).

   f) The <language name> included in *DCMS*.

   g) The <parameter style> included in *DCMS* (if any).

   h) The <returns data type> contained in *POVMS*.

   i) The <result cast from type> included in *DCMS* (if any).

   j) The locator indication contained in the <returns clause> included in the *DCMS*.

   k) The determinism indication included in *DCMS*.

   l) The SQL-data access indication included in *DCMS* (if any).

   m) The indication included in *DCMS* (if at all), whether the method should be invoked if any argument is the null value.

2) *DOMS* is added to *STDS*.

3) Let *N* be the number of table descriptors that include the user-defined type name of a subtype of *D*.

   For *i* varying from 1 (one) to *N*:

   a) Let TN_i be the <table name> included in the *i*-th such table descriptor.

b) Let *M* be the number of table/method privilege descriptors that specify TN_i and the <specific method name> contained in *COMS*. For *j* varying from 1 (one) to *M*:

   i) Let $TMPD_j$ be the *j*-th such table/method privilege descriptor.

   ii) A new table/method privilege descriptor is created that specifies TN_i, the same action, grantor, and grantee, and the same grantability, and the <specific method name> contained in *OVMS*.

   iii) $TMPD_j$ is deleted.

Conformance Rules

None.

11.48 <drop method specification>

Function

Remove a method specification from a user-defined type.

Format

```
<drop method specification> ::=
  DROP <specific method specification designator> RESTRICT

<specific method specification designator> ::=
  [ INSTANCE | STATIC | CONSTRUCTOR ]
      METHOD <method name> <data type list>
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> *DN* immediately contained in the <alter type statement> containing the <drop method specification> *DORMS*. Let *DSN* be the explicit or implicit <schema name> of *DN*. Let *SMSD* be the <specific method specification designator> immediately contained in *DORMS*.

2) If *SMSD* immediately contains a <specific method name> *SMN*, then:

   a) If *SMN* contains a <schema name>, then that <schema name> shall be equivalent to *DSN*. Otherwise, the <schema name> *DSN* is implicit.

   b) The descriptor of *D* shall include a method specification descriptor *DOOMS* whose specific method name is equivalent to *SMN*.

   c) Let *PDL* be the augmented parameter list included in *DOOMS*.

   d) Let *MN* be the <method name> included in *DOOMS*.

3) If *SMSD* immediately contains a <method name> *ME*, then:

   a) If none of INSTANCE, STATIC, or CONSTRUCTOR is immediately contained in *SMSD*, then INSTANCE is implicit.

   b) The descriptor of *D* shall include a method specification descriptor *DOOMS* whose method name *MN* is equivalent to *ME*.

   c) If *SMSD* immediately contains a <data type list> *DTL*, then

      Case:

      i) If STATIC is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes:

         1) An indication that the method specification is STATIC.

         2) An indication that the method specification is original.

3) An augmented parameter list *PDL* such that the declared type of its *i*-th parameter, for all *i*, is identical to the *i*-th declared type in *DTL*.

ii) If CONSTRUCTOR is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes:

1) An indication that the method specification is CONSTRUCTOR.

2) An indication that the method specification is original.

3) An augmented parameter list *PDL* such that the declared type of its *i*-th parameter, for all *i* > 1 (one), is identical to the (*i*–1)-th declared type in *DTL* and the declared type of the first parameter of *PDL* is identical to *DN*.

iii) Otherwise, the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* for which:

1) If *DOOMS* includes an indication that the method specification is original, then *DOOMS* shall not include an indication that the method specification is either STATIC or CONSTRUCTOR.

2) *DOOMS* includes an augmented parameter list *PDL* such that the declared type of its *i*-th parameter, for all *i* > 1 (one), is identical to the (*i*–1)-th declared type in *DTL* and the declared type of the first parameter of *PDL* is identical to *DN*.

d) If *SMSD* does not immediately contain a <data type list>, then

Case:

i) If STATIC is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes an indication that the method specification is both original and STATIC.

ii) If CONSTRUCTOR is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes an indications that the method specification is both original and CONSTRUCTOR.

iii) Otherwise, the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* for which if *DOOMS* includes an indication that the method specification is original, then *DOOMS* shall not include an indication that the method specification is either STATIC or CONSTRUCTOR.

4) Case:

a) If *DOOMS* includes an indication that the method specification is original, then

Case:

i) If *DOOMS* includes an indication that the method specification specified STATIC, then there shall be no SQL-invoked function *F* that satisfies all of the following conditions:

1) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

2) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

3) The declared type of the first SQL parameter of *F* is *D*.

4) The declared type of the *i*-th element of *PDL*, 1 (one) ≤ *i* ≤ *N*, is compatible with the declared type of SQL parameter P_i of *F*.

5) *F* is an SQL-invoked method.

6) *F* includes an indication that STATIC is specified.

ii) If *DOOMS* includes an indication that the method specification specified CONSTRUCTOR, then there shall be no SQL-invoked function *F* that satisfies all of the following conditions:

1) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

2) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

3) The declared type of the first SQL parameter of *F* is *D*.

4) The declared type of the *i*-th element of *PDL*, 2 ≤ *i* ≤ *N*, is compatible with the declared type of SQL parameter P_i of *F*.

5) *F* is an SQL-invoked method.

6) *F* includes an indication that CONSTRUCTOR is specified.

iii) Otherwise:

1) There shall be no proper subtype *PSBD* of *D* whose descriptor includes the descriptor *DOVMS* of an overriding method specification such that all of the following is true:

A) *MN* and the <method name> included in *DOVMS* have equivalent <qualified identifier>s.

B) If *N* is the number of elements in *PDL*, then the augmented SQL parameter declaration list *APDL* included in *DOVMS* has *N* SQL parameters.

C) *PSBD* is the declared type the first SQL parameter of *APDL*.

D) The declared type of the *i*-th element of *PDL*, 2 ≤ *i* ≤ *N*, is compatible with the declared type of SQL parameter P_i of *APDL*.

2) There shall be no SQL-invoked function *F* that satisfies all of the following conditions:

A) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

B) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

C) The declared type of the first SQL parameter of *F* is *D*.

D) The declared type of the *i*-th element of *PDL*, 2 ≤ *i* ≤ *N*, is compatible with the declared type of SQL parameter P_i of *F*.

E) *F* is an SQL-invoked method.

F) *F* does not include an indication that either STATIC or CONSTRUCTOR is specified.

b) Otherwise, there shall be no SQL-invoked function *F* that satisfies all of the following conditions:

i) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

ii) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

iii) The declared type of the first SQL parameter of *F* is *D*.

iv) The declared type of the *i*-th element of *PDL*, $2 \le i \le N$, is compatible with the declared type of SQL parameter P_i of *F*.

v) *F* is an SQL-invoked method.

vi) *F* does not include an indication that either STATIC or CONSTRUCTOR is specified.

Access Rules

None.

General Rules

1) Let *STDS* be the descriptor of *D*.

2) *DOOMS* is removed from *STDS*.

3) *DOOMS* is destroyed.

Conformance Rules

None.

11.49 <drop data type statement>

This Subclause is modified by Subclause 9.23, "<drop data type statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 9.7, "<drop data type statement>", in ISO/IEC 9075-13.

Function

Destroy a user-defined type.

Format

```
<drop data type statement> ::=
  DROP TYPE <schema-resolved user-defined type name> <drop behavior>
```

Syntax Rules

1) Let *DN* be the <schema-resolved user-defined type name> and let *D* be the data type identified by *DN*. Let *SD* be any supertype of *D*.

2) Let *RD* be the reference type whose referenced type is *D*. Let *SRD* be any supertype of *RD*. Let *AD* be any collection type whose element type is *D*. Let *SAD* be any collection type whose element type is a supertype of *D* or *RD*.

3) The schema identified by the explicit or implicit schema name of *DN* shall include the descriptor of *D*.

4) If RESTRICT is specified, then:

   a) The declared type of no column, field, or attribute whose descriptor is not included in the descriptor of *D* shall be *SRD* or *SAD*.

   b) The declared type of no column, attribute, or field shall be based on *D*.

   c) *D* shall have no proper subtypes.

   d) *D* shall not be the structured type of a referenceable table.

   e) The transform descriptor included in the user-defined type descriptor of *D* shall include an empty list of transform groups.

   f) *D*, *RD*, and *AD* shall not be referenced in any of the following:

      i) The <query expression> of any view descriptor.

      ii) The <search condition> of any constraint descriptor.

      iii) A trigger action of any trigger descriptor.

      iv) A user-defined cast descriptor.

      v) A user-defined type descriptor other than that of *D* itself.

   g) There shall be no SQL-invoked routine that is not dependent on *D* and whose routine descriptor includes the descriptor of *D*, *RD*, or *AD*, or whose SQL routine body references *D*, *RD*, or *AD*.

h) Let *R* be any SQL-invoked routine that is dependent on *D* and whose routine descriptor includes the descriptor of *D* or *RD*.

i) *R* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any of the following:

1) The SQL routine body of any routine descriptor.

2) The <query expression> of any view descriptor.

3) The <search condition> of any constraint descriptor.

4) The trigger action of any trigger descriptor.

ii) The specific name of *R* shall not be included in any user-defined cast descriptor.

iii) *R* shall not be the ordering function included in the descriptor of any user-defined type.

NOTE 331 — If CASCADE is specified, then such referenced objects will be dropped by the execution of the <revoke statement> specified in the General Rules of this Subclause.

NOTE 332 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.27, "SQL-invoked routines".

NOTE 333 — The notion of one data type being based on another data type is defined in Subclause 4.1, "Data types".

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *D*.

General Rules

1) Let *SN* be the <specific name> of any <SQL-invoked routine> that references *D*, *RD*, or *AD* or whose routine descriptor includes the descriptor of *D*, *RD*, or *AD* and that is not dependent on *D*. The following <drop routine statement> is effectively executed for each such <SQL-invoked routine> without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

NOTE 334 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.27, "SQL-invoked routines".

2) The following <drop transform statement> is effectively executed without further Access Rule checking:

```
DROP TRANSFORM ALL FOR DN CASCADE
```

NOTE 335 — This Rule should have no effect, since any external routine that depends on the transform being dropped also depends on the data type for which the transform is defined and hence should have already been dropped because of General Rule 1).

3) Let *UDCD* be the user-defined cast descriptor that references *DN* as the source data type. Let *TD* be the target data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( DN AS TD ) CASCADE
```

4) Let *UDCD* be the user-defined cast descriptor that references *DN* as the target data type. Let *SD* be the source data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( SD AS DN ) CASCADE
```

5) Let *UDCD* be the user-defined cast descriptor that references the reference type whose referenced type is *DN* as the source data type. Let *TD* be the target data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( REF (DN) AS TD ) CASCADE
```

6) Let *UDCD* be the user-defined cast descriptor that references the reference type whose referenced type is *DN* as the target data type. Let *SD* be the source data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( SD AS REF (DN) ) CASCADE
```

7) For every privilege descriptor that references *D*, the following <revoke statement> is effectively executed:

```
REVOKE PRIV ON TYPE D FROM GRANTEE CASCADE
```

where *PRIV* and *GRANTEE* are respectively the action and grantee in the privilege descriptor.

8) The descriptor of every SQL-invoked routine that is said to be dependent on *D* is destroyed.

NOTE 336 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.27, "SQL-invoked routines".

9) The descriptor of *D* is destroyed.

Conformance Rules

1) Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop data type statement> that contains a <drop behavior> that contains CASCADE.

11.50 <SQL-invoked routine>

This Subclause is modified by Subclause 9.24, "<SQL-invoked routine>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.10, "<SQL-invoked routine>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 9.8, "<SQL-invoked routine>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.7, "<SQL-invoked routine>", in ISO/IEC 9075-14.

Function

Define an SQL-invoked routine.

Format

```
<SQL-invoked routine> ::=
  <schema routine>

<schema routine> ::=
    <schema procedure>
  | <schema function>

<schema procedure> ::=
  CREATE <SQL-invoked procedure>

<schema function> ::=
  CREATE <SQL-invoked function>

<SQL-invoked procedure> ::=
  PROCEDURE <schema qualified routine name> <SQL parameter declaration list>
      <routine characteristics>
      <routine body>

<SQL-invoked function> ::=
  { <function specification> | <method specification designator> } <routine body>

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

<SQL parameter declaration> ::=
  [ <parameter mode> ] [ <SQL parameter name> ] <parameter type> [ RESULT ]

<parameter mode> ::=
    IN
  | OUT
  | INOUT

<parameter type> ::=
  <data type> [ <locator indication> ]

<locator indication> ::=
  AS LOCATOR

<function specification> ::=
  FUNCTION <schema qualified routine name> <SQL parameter declaration list>
      <returns clause>
```

```
      <routine characteristics>
      [ <dispatch clause> ]

<method specification designator> ::=
    SPECIFIC METHOD <specific method name>
  | [ INSTANCE | STATIC | CONSTRUCTOR ]
      METHOD <method name> <SQL parameter declaration list>
      [ <returns clause> ]
      FOR <schema-resolved user-defined type name>

<routine characteristics> ::=
  [ <routine characteristic>... ]

<routine characteristic> ::=
    <language clause>
  | <parameter style clause>
  | SPECIFIC <specific name>
  | <deterministic characteristic>
  | <SQL-data access indication>
  | <null-call clause>
  | <returned result sets characteristic>
  | <savepoint level indication>

<savepoint level indication> ::=
    NEW SAVEPOINT LEVEL
  | OLD SAVEPOINT LEVEL

<returned result sets characteristic> ::=
  DYNAMIC RESULT SETS <maximum returned result sets>

<parameter style clause> ::=
  PARAMETER STYLE <parameter style>

<dispatch clause> ::=
  STATIC DISPATCH

<returns clause> ::=
  RETURNS <returns type>

<returns type> ::=
    <returns data type> [ <result cast> ]
  | <returns table type>

<returns table type> ::=
  TABLE <table function column list>

<table function column list> ::=
  <left paren> <table function column list element>
      [ { <comma> <table function column list element> }... ] <right paren>

<table function column list element> ::=
  <column name> <data type>

<result cast> ::=
  CAST FROM <result cast from type>

<result cast from type> ::=
  <data type> [ <locator indication> ]

<returns data type> ::=
```

```
  <data type> [ <locator indication> ]

<routine body> ::=
    <SQL routine spec>
  | <external body reference>

<SQL routine spec> ::=
  [ <rights clause> ] <SQL routine body>

<rights clause> ::=
    SQL SECURITY INVOKER
  | SQL SECURITY DEFINER

<SQL routine body> ::=
  <SQL procedure statement>

<external body reference> ::=
  EXTERNAL [ NAME <external routine name> ]
      [ <parameter style clause> ]
      [ <transform group specification> ]
      [ <external security clause> ]

<external security clause> ::=
    EXTERNAL SECURITY DEFINER
  | EXTERNAL SECURITY INVOKER
  | EXTERNAL SECURITY IMPLEMENTATION DEFINED

<parameter style> ::=
    SQL
  | GENERAL

<deterministic characteristic> ::=
    DETERMINISTIC
  | NOT DETERMINISTIC

<SQL-data access indication> ::=
    NO SQL
  | CONTAINS SQL
  | READS SQL DATA
  | MODIFIES SQL DATA

<null-call clause> ::=
    RETURNS NULL ON NULL INPUT
  | CALLED ON NULL INPUT

<maximum returned result sets> ::=
  <unsigned integer>

<transform group specification> ::=
  TRANSFORM GROUP { <single group specification> | <multiple group specification> }

<single group specification> ::=
  <group name>

<multiple group specification> ::=
  <group specification> [ { <comma> <group specification> }... ]

<group specification> ::=
  <group name> FOR TYPE <path-resolved user-defined type name>
```

Syntax Rules

1) An <SQL-invoked routine> specifies an *SQL-invoked routine*. Let *R* be the SQL-invoked routine specified by <SQL-invoked routine>.

2) If <SQL-invoked routine> immediately contains <schema routine>, then the SQL-invoked routine identified by <schema qualified routine name> is a *schema-level routine*.

3) An <SQL-invoked routine> specified as an <SQL-invoked procedure> is called an *SQL-invoked procedure*; an <SQL-invoked routine> specified as an <SQL-invoked function> is called an *SQL-invoked function*. An <SQL-invoked function> that specifies a <method specification designator> is further called an *SQL-invoked method*. An SQL-invoked method that specifies STATIC is called a *static SQL-invoked method*. An SQL-invoked method that specifies CONSTRUCTOR is called an *SQL-invoked constructor method*.

4) If <returns type> *RST*specifies TABLE, then let *TCL* be the <table function column list> contained in <returns table type>.

   a) For every <column name> *CN* contained in *TCL*, *CN* shall not be equivalent to any other <column name> contained in *TCL*.

   b) *RST* is equivalent to the <returns type>

      ```
      ROW TCL MULTISET
      ```

5) If <SQL-invoked routine> specifies an SQL-invoked method, then

   Case:

   a) If a <specific method name> *SMN* is specified, then:

      i) Case:

         1) If *SMN* does not contain <schema name>, then

            Case:

            A) If the <SQL-invoked routine> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

            B) Otherwise, the <schema name> that is specified or implicit for the <SQL-client module definition> is implicit.

         2) Otherwise, if <SQL-invoked routine> is contained in a <schema definition> then the <schema name> contained in *SMN* shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

      ii) Let *S* be the schema identified by the implicit or explicit <schema name> of *SMN*.

      iii) There shall exist a method specification descriptor *DMS* included in the descriptor of a user-defined type *UDT* included in *S*, whose <specific method name> is *SMN*.

      iv) Let *MN* be the number of SQL parameters in the unaugmented SQL parameter declaration list in *DMS*. *MN* is the number of SQL parameters in the unaugmented SQL parameter declaration list of *R*.

      v) If *DMS* includes <result cast> *RC*, then *RC* is the <result cast> of *R*.

vi) Let *SPN* be the <specific method name> in *DMS*. *SPN* is the <specific name> of *R*.

vii) Let *NPL* be the augmented SQL parameter declaration list of *DMS*. *NPL* is the augmented SQL parameter declaration list of *R*.

viii) Let *RN* be *SN*.<method name>, where *SN* is the <schema name> of the schema that includes the descriptor of *UDT*.

b) Otherwise:

i) Let *UDTN* be the <schema-resolved user-defined type name> immediately contained in <method specification designator>. Let *UDT* be the user-defined type identified by *UDTN*.

ii) There shall exist a method specification descriptor *DMS* in the descriptor of *UDT* such that the <method name> of *DMS* is equivalent to the <method name>, *DMS* indicates STATIC if and only if the <method specification designator> specifies STATIC, *DMS* indicates CONSTRUCTOR if and only if the <method specification designator> specifies CONSTRUCTOR, and the declared type of every SQL parameter in the unaugmented SQL parameter declaration list in *DMS* is compatible with the declared type of the corresponding SQL parameter in the <SQL parameter declaration list> contained in the <method specification designator>. *DMS* identifies the corresponding method specification of the <method specification designator>.

iii) Let *MN* be the number of SQL parameters in the unaugmented SQL parameter declaration list in *DMS*.

iv) Let $PCOMS_i$, 1 (one) $\le i \le MN$, be the *i*-th SQL parameter in the unaugmented SQL parameter declaration list of *DMS*. Let $POVMS_i$, 1 (one) $\le i \le MN$, be the *i*-th SQL parameter contained in <method specification designator>.

v) For *i* varying from 1 (one) to *MN*, the <SQL parameter name>s contained in $PCOMS_i$ and $POVMS_i$ shall be equivalent.

vi) Let $PDMS_i$, 1 (one) $\le i \le MN$, be the declared type of the *i*-th SQL parameter in the unaugmented SQL parameter declaration list in *DMS*. Let PSM_i be the declared type of the *i*-th SQL parameter contained in <method specification designator>.

vii) With *i* ranging from 1 (one) to *MN*, the Syntax Rules of Subclause 9.18, "Data type identity", are applied with $PDMS_i$ and PSM_i.

viii) Case:

1) If <returns clause> is specified, then let *RT* be the <returns data type> of *R*. Let *RDMS* be the <returns data type> in *DMS*. The Syntax Rules of Subclause 9.18, "Data type identity", are applied with *RT* and *RDMS*.

2) Otherwise, let *RDMS* be the <returns data type> of *R*.

ix) If *DMS* includes <result cast> *RC*, then

Case:

1) If <returns clause> is specified, then <returns clause> shall contain <result cast>. Let *RDCT* be the <data type> specified in *RC*. Let *RCT* be the <data type> specified in the <result cast>

contained in <returns clause>. The Syntax Rules of Subclause 9.18, "Data type identity", are applied with *RDCT* and *RCT*.

2) Otherwise, *RC* is the <result cast> of *R*.

x) Let *SPN* be the <specific method name> in *DMS*. *SPN* is the <specific name> of *R*.

xi) Let *NPL* be the augmented SQL parameter declaration list of *DMS*.

xii) Let *RN* be *SN*.<method name>, where *SN* is the <schema name> of the schema that includes the descriptor of *UDT*.

6) If <SQL-invoked routine> specifies an SQL-invoked procedure or an SQL-invoked regular function, then:

a) <routine characteristics> shall contain at most one <language clause>, at most one <parameter style clause>, at most one <specific name>, at most one <deterministic characteristic>, at most one <SQL-data access indication>, at most one <null-call clause>, and at most one <returned result sets characteristic>.

b) <parameter style clause> shall not be specified both in <routine characteristics> and in <external body reference>.

c) The <routine characteristics> of a <function specification> shall not contain a <returned result sets characteristic>.

d) If the SQL-invoked routine is an SQL-invoked procedure and <returned result sets characteristic> is not specified, then DYNAMIC RESULT SETS 0 (zero) is implicit.

e) If <deterministic characteristic> is not specified, then NOT DETERMINISTIC is implicit.

f) Case:

i) If PROCEDURE is specified, then:

1) <null-call clause> shall not be specified.

2) <routine characteristics> shall not contain more than one <savepoint level indication>.

ii) Otherwise, if <null-call clause> is not specified, then CALLED ON NULL INPUT is implicit.

g) <SQL-data access indication> shall be specified.

h) If <language clause> is not specified, then LANGUAGE SQL is implicit.

i) An <SQL-invoked routine> that specifies or implies LANGUAGE SQL is called an *SQL routine*; an <SQL-invoked routine> that does not specify LANGUAGE SQL is called an *external routine*.

j) If <savepoint level indication> is specified, then PROCEDURE shall be specified.

k) If PROCEDURE is specified and <savepoint level indication> is not specified, then OLD SAVEPOINT LEVEL is implicit.

l) If NEW SAVEPOINT LEVEL is specified, then MODIFIES SQL DATA shall be specified.

m) If *R* is an SQL routine, then:

i) The <returns clause> shall not specify a <result cast>.

ii) <SQL-data access indication> shall not specify NO SQL.

iii) <parameter style clause> shall not be specified.

n) An array-returning external function is an SQL-invoked function that is an external routine and that satisfies one of the following conditions:

i) A <result cast from type> is specified that simply contains an <array type> and does not contain a <locator indication>.

ii) A <result cast from type> is not specified and <returns data type> simply contains an <array type> and does not contain a <locator indication>.

o) A multiset-returning external function is an SQL-invoked function that is an external routine and that satisfies one of the following conditions:

i) A <result cast from type> is specified that simply contains a <multiset type> and does not contain a <locator indication>.

ii) A <result cast from type> is not specified and <returns data type> simply contains a <multiset type> and does not contain a <locator indication>.

p) Let *RN* be the <schema qualified routine name> of *R*.

q) If <SQL-invoked routine> is contained in a <schema definition> and *RN* contains a <schema name> *SN*, then *SN* shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>. Let *S* be the SQL-schema identified by *SN*.

r) Case:

i) If *R* is an SQL-invoked regular function and the <SQL parameter declaration list> contains an <SQL parameter declaration> that specifies a <data type> that is one of:

1) A user-defined type.

2) A collection type whose element type is a user-defined type.

3) A collection type whose element type is a reference type.

4) A reference type.

then <dispatch clause> shall be specified.

ii) Otherwise, <dispatch clause> shall not be specified.

s) If <specific name> is not specified, then an implementation-dependent <specific name> whose <schema name> is the equivalent to the <schema name> of *S* is implicit.

t) If <specific name> contains a <schema name>, then that <schema name> shall be equivalent to the <schema name> of *S*. If <specific name> does not contain a <schema name>, then the <schema name> of *S* is implicit.

u) The schema identified by the explicit or implicit <schema name> of the <specific name> shall not include a routine descriptor whose specific name is equivalent to <specific name> or a user-defined type descriptor that includes a method specification descriptor whose specific name is equivalent to <specific name>.

v) If <returns data type> *RT* simply contains <locator indication>, then:

i) *R* shall be an external routine.

ii) *RT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

iii) <result cast> shall not be specified.

w) If <result cast from type> *RCT* simply contains <locator indication>, then:

i) *R* shall be an external routine.

ii) *RCT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

x) If *R* is an external routine, then:

i) If <parameter style> is not specified, then PARAMETER STYLE SQL is implicit.

ii) If *R* is an array-returning external function or a multiset-returning external function, then PARAMETER STYLE SQL shall be either specified or implied.

iii) Case:

1) If <transform group specification> is not specified, then a <multiple group specification> with a <group specification> *GS* for each <SQL parameter declaration> contained in <SQL parameter declaration list> whose <parameter type> *UDT* identifies a user-defined type with no <locator indication> is implicit. The <group name> of *GS* is implementation-defined and its <path-resolved user-defined type name> is *UDT*.

2) If <single group specification> with a <group name> *GN* is specified, then <transform group specification> is equivalent to a <transform group specification> that contains a <multiple group specification> that contains a <group specification> *GS* for each <SQL parameter declaration> contained in <SQL parameter declaration list> whose <parameter type> *UDT* identifies a user-defined type with no <locator indication>. The <group name> of *GS* is *GN* and its <path-resolved user-defined type name> is *UDT*.

3) Otherwise, <multiple group specification> is extended with a <group specification> *GS* for each <SQL parameter declaration> contained in <SQL parameter declaration list> whose <parameter type> *UDT* identifies a user-defined type with no <locator indication> and no equivalent of *UDT* is contained in any <group specification> contained in <multiple group specification>. The <group name> of *GS* is implementation-defined and its <path-resolved user-defined type name> is *UDT*.

iv) If a <result cast> is specified, then let *V* be some value of the <data type> specified in the <result cast> and let *RT* be the <returns data type>. The following shall be valid according to the Syntax Rules of Subclause 6.12, "<cast specification>":

```
CAST ( V AS RT )
```

y) Let *NPL* be the <SQL parameter declaration list> contained in the <SQL-invoked routine>.

7) *NPL* specifies the list of SQL parameters of *R*. Each SQL parameter of *R* is specified by an <SQL parameter declaration>. If <SQL parameter name> is specified, then that SQL parameter of *R* is identified by an SQL parameter name.

8) *NPL* shall specify at most one <SQL parameter declaration> that specifies RESULT.

9) If *R* is an SQL-invoked function, then no <SQL parameter declaration> in *NPL* shall contain a <parameter mode>.

10) If *R* is an SQL routine, then every <SQL parameter declaration> in *NPL* shall contain an <SQL parameter name>.

11) No two <SQL parameter name>s contained in *NPL* shall be equivalent.

12) Let *N* and *PN* be the number of <SQL parameter declaration>s contained in *NPL*. For every <SQL parameter declaration> PD_i, 1 (one) ≤ i ≤ N:

    a) <parameter type> PT_i immediately contained in PD_i shall not specify ROW.

    b) If PT_i simply contains <locator indication>, then:

        i) *R* shall be an external routine.

        ii) PT_i shall specify either binary large object type, character large object type, array type, multiset type, or user-defined type.

    c) If PD_i immediately contains RESULT, then:

        i) *R* shall be an SQL-invoked function.

        ii) PT_i shall specify a structured type *ST*. Let *STN* be the <user-defined type name> that identifies *ST*.

        iii) The <returns data type> shall specify *STN*.

        iv) *R* is a type-preserving function and PD_i specifies the result SQL parameter of *R*.

    d) If PD_i does not contain a <parameter mode>, then a <parameter mode> that specifies IN is implicit.

    e) Let P_i be the *i*-th SQL parameter.

        Case:

        i) If the <parameter mode> specifies IN, then P_i is an input SQL parameter.

        ii) If the <parameter mode> specifies OUT, then P_i is an output SQL parameter.

        iii) If the <parameter mode> specifies INOUT, then P_i is both an input SQL parameter and an output SQL parameter.

13) The scope of *RN* is the <routine body> of *R*.

14) The scope of an <SQL parameter name> contained in *NPL* is the <routine body> *RB* of the <SQL-invoked procedure> or <SQL-invoked function> that contains *NPL*.

15) An <SQL-invoked routine> shall not contain a <host parameter name>, a <dynamic parameter specification>, or an <embedded variable name>.

16) Case:

    a) If *R* is an SQL-invoked procedure, then *S* shall not include another SQL-invoked procedure whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

b) Otherwise:

i) Case:

1) If *R* is a static SQL-invoked method, then let *SCR* be the set containing every static SQL-invoked method of type *UDT*, including *R*, whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

2) If *R* is an SQL-invoked constructor method, then let *SCR* be the set containing every SQL-invoked constructor method of type *UDT*, including *R*, whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

3) Otherwise, let *SCR* be the set containing every SQL-invoked function in *S* that is neither a static SQL-invoked method nor an SQL-invoked constructor method, including *R*, whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

ii) Let *AL* be an <SQL argument list> constructed from a list of arbitrarily-selected values in which the declared type of every value A_i in *AL* is compatible with the declared type of the corresponding SQL parameter P_i of *R*.

iii) For every A_i, eliminate from *SCR* every SQL-invoked routine *SIR* for which the type designator of the declared type of the SQL parameter P_i of *SIR* is not in the type precedence list of the declared type of A_i.

iv) Let *SR* be the set of subject routines defined by applying the Syntax Rules of Subclause 9.4, "Subject routine determination", with the set of SQL-invoked routines as *SCR* and <SQL argument list> as *AL*. There shall be exactly one subject routine in *SR*.

17) If *R* is an SQL-invoked method but not a static SQL-invoked method, then the first SQL parameter of *NPL* is called the *subject parameter* of *R*.

18) If *R* is an SQL-invoked regular function *F* whose first SQL parameter has a declared type that is a user-defined type, then:

a) Let *UDT* be the declared type of the first SQL parameter of *F*.

b) Let *DMS* be a method specification descriptor of an instance method in the descriptor of *UDT* such that:

i) The <schema qualified routine name> of *F* and the <routine name> of *DMS* have equivalent <qualified identifier>s.

ii) *F* and the augmented SQL parameter declaration list of *DMS* have the same number of SQL parameters.

c) Let $PDMS_i$, 1 (one) $\le i \le PN$, be the declared type of the *i*-th SQL parameter in the unaugmented SQL parameter declaration list in *DMS* and let PMS_i be the declared type of the *i*-th SQL parameter contained in <function specification>.

d) One of the following conditions shall be false:

i) The declared type of $PDMS_i$, 1 (one) $\le i \le N$ is compatible with the declared type of SQL parameter PMS_{i+1}.

ii) *UDT* is a subtype or a supertype of the declared type of PMS_1.

19) If *R* is an SQL routine, then:

a) <SQL routine spec> shall be specified.

b) If <rights clause> is not specified, then SQL SECURITY DEFINER is implicit.

c) If READS SQL DATA is specified, then it is implementation-defined whether the <SQL routine body> shall not contain an <SQL procedure statement> *S* that satisfies at least one of the following:

i) *S* is an <SQL data change statement>.

ii) *S* contains a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

iii) *S* contains an <SQL procedure statement> that is an <SQL data change statement>.

d) If CONTAINS SQL is specified, then it is implementation-defined whether the <SQL routine body> shall not contain an <SQL procedure statement> *S* that satisfies at least one of the following:

i) *S* is an <SQL data statement> other than <free locator statement> and <hold locator statement>.

ii) *S* contains a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data or possibly reads SQL-data.

iii) *S* contains an <SQL procedure statement> that is an <SQL data statement> other than <free locator statement> and <hold locator statement>.

e) If DETERMINISTIC is specified, then it is implementation-defined whether the <SQL routine body> shall not contain an <SQL procedure statement> that is possibly non-deterministic.

f) It is implementation-defined whether the <SQL routine body> shall not contain an <SQL connection statement>, an <SQL schema statement>, an <SQL dynamic statement>, or an <SQL transaction statement> other than a <savepoint statement>, <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>.

NOTE 337 — Conforming SQL language shall not contain an <SQL connection statement> or an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>, but an implementation is not required to treat this as a syntax error.

g) An <SQL routine body> shall not immediately contain an <SQL procedure statement> that simply contains a <schema definition>.

20) If *R* is an external routine, then:

a) <SQL routine spec> shall not be specified.

b) If <external security clause> is not specified, then EXTERNAL SECURITY IMPLEMENTATION DEFINED is implicit.

c) If an <external routine name> is not specified, then an <external routine name> that is equivalent to the <qualified identifier> of *R* is implicit.

d) If PARAMETER STYLE SQL is specified, then:

i) Case:

1) If *R* is an array-returning external function or a multiset-returning external function with the element type being a row type, then let *FRN* be the degree of the element type.

2) Otherwise, let *FRN* be 1 (one).

ii) If *R* is an array-returning external function or a multiset-returning external function, then let *AREF* be *FRN*+6. Otherwise, let *AREF* be *FRN*+4.

iii) If *R* is an SQL-invoked function, then let the *effective SQL parameter list* be a list of *PN+FRN+N+AREF* SQL parameters, as follows:

1) For *i* ranging from 1 (one) to *PN*, the *i*-th effective SQL parameter list entry is defined as follows.

Case:

A) If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> contains <locator indication>, then the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by INTEGER.

B) If the <parameter type> T_i immediately contained in the *i*-th <SQL parameter declaration> is a <path-resolved user-defined type name> without a <locator indication>, then:

I) Case:

1) If *R* is an SQL-invoked method that is an overriding method, then the Syntax Rules of Subclause 9.20, "Determination of a from-sql function for an overriding method", are applied with *R* and *i* as *ROUTINE* and *POSITION*, respectively. There shall be an applicable from-sql function FSF_i.

2) Otherwise, the Syntax Rules of Subclause 9.19, "Determination of a from-sql function", are applied with the data type identified by T_i, and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable from-sql function FSF_i.

II) FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

III) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

C) Otherwise, the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration>.

2) Case:

A) If *FRN* is 1 (one), then effective SQL parameter list entry *PN+FRN* has <parameter mode> OUT; its <parameter type> *PT* is defined as follows:

I) If <result cast> is specified, then let *RT* be <result cast from type>; otherwise, let *RT* be <returns data type>.

II) Case:

1) If *RT* simply contains <locator indication>, then *PT* is INTEGER.

2) If *RT* specifies a <path-resolved user-defined type name> without a <locator indication>, then:

a) Case:

i) If *R* is an SQL-invoked method that is an overriding method, then the Syntax Rules of Subclause 9.22, "Determination of a to-sql function for an overriding method", are applied with *R* as *ROUTINE*. There shall be an applicable to-sql function *TSF*.

ii) Otherwise, the Syntax Rules of Subclause 9.21, "Determination of a to-sql function", are applied with the data type identified by *RT* and the <group name> contained in the <group specification> that contains *RT* as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function *TSF*.

b) *TSF* is called the *to-sql function* associated with the result.

c) Case:

i) If *TSF* is an SQL-invoked method, then *PT* is the <parameter type> of the second SQL parameter of *TSF*.

ii) Otherwise, *PT* is the <parameter type> of the first SQL parameter of *TSF*.

3) If *R* is an array-returning external function or a multiset-returning external function, then let *PT* be the element type of *RT*.

4) Otherwise, *PT* is *RT*.

B) Otherwise, for *i* ranging from *PN*+1 to *PN*+*FRN*, the *i*-th effective SQL parameter list entry is defined as follows.

Case:

I) Its <parameter mode> is OUT.

II) Let $RFT_{i\text{-}PN}$ be the data type of the *i*-*PN*-th field of the element type of the <returns data type>. The <parameter type> PT_i of the *i*-th effective SQL parameter list entry is determined as follows:

1) If $RFT_{i\text{-}PN}$ specifies a <path-resolved user-defined type name>, then:

a) Case:

i) If *R* is an SQL-invoked method that is an overriding method, then the Syntax Rules of Subclause 9.22, "Determination of a to-sql function for an overriding method", are applied with *R* as *ROUTINE*. There shall be an applicable to-sql function *TSF*.

ii) Otherwise, the Syntax Rules of Subclause 9.21, "Determination of a to-sql function", are applied with the data type identified by $RFT_{i\text{-}PN}$ and the <group name> contained in the <group specification> that contains $RFT_{i\text{-}PN}$ as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function *TSF*.

b) *TSF* is called the *to-sql function* associated with $RFT_{i\text{-}PN}$.

c) Case:

i) If *TSF* is an SQL-invoked method, then PT_i is the <parameter type> of the second SQL parameter of *TSF*.

ii) Otherwise, PT_i is the <parameter type> of the first SQL parameter of *TSF*.

2) Otherwise, PT_i is $RFT_{i\text{-}PN}$.

3) Effective SQL parameter list entries (*PN*+*FRN*)+1 to (*PN*+*FRN*)+*N*+*FRN* are *N*+*FRN* occurrences of SQL parameters of an implementation-defined <data type> that is an exact numeric type with scale 0 (zero). For *i* ranging from (*PN*+*FRN*)+1 to (*PN*+*FRN*)+*N*+*FRN*, the <parameter mode> for the *i*-th such effective SQL parameter is the same as that of the *i*–*FRN*–*PN*-th effective SQL parameter.

4) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+1 is an SQL parameter of a <data type> that is character string of length 5 and the character set specified for SQLSTATE values, with <parameter mode> INOUT.

NOTE 338 — The character set specified for SQLSTATE values is defined in Subclause 24.1, "SQLSTATE".

5) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+2 is an SQL parameter of a <data type> that is character string of implementation-defined length and character set SQL_TEXT with <parameter mode> IN.

6) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+3 is an SQL parameter of a <data type> that is character string of implementation-defined length and character set SQL_TEXT with <parameter mode> IN.

7) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+4 is an SQL parameter of a <data type> that is character string of implementation-defined length and character set SQL_TEXT with <parameter mode> INOUT.

8) If *R* is an array-returning external function or a multiset-returning external function, then:

A) Effective SQL parameter type list entry (*PN*+*FRN*)+(*N*+*FRN*)+5 is an SQL parameter whose <data type> is character string of implementation-defined length and character set SQL_TEXT with <parameter mode> INOUT.

B) Effective SQL parameter type list entry (*PN*+*FRN*)+(*N*+*FRN*)+6 is an SQL parameter whose <data type> is an exact numeric type with scale 0 (zero) and with <parameter mode> IN.

iv) If *R* is an SQL-invoked procedure, then let the *effective SQL parameter list* be a list of *PN*+*N*+4 SQL parameters, as follows:

1) For *i* ranging from 1 (one) to *PN*, the *i*-th effective SQL parameter list entry is defined as follows.

Case:

A) If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> simply contains <locator indication>, then the *i*-th effective SQL parameter list entry

is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by INTEGER.

B) If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> is a <path-resolved user-defined type name> without a <locator indication>, then:

I) Case:

1) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is IN, then:

a) The Syntax Rules of Subclause 9.19, "Determination of a from-sql function", are applied with the data type identified by T_i and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable from-sql function FSF_i. FSF_i is called the *from-sql function* associated with the *i*-th SQL parameter.

b) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

2) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is OUT, then:

a) The Syntax Rules of Subclause 9.21, "Determination of a to-sql function", are applied with the data type identified by T_i and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function TSF_i. TSF_i is called the *to-sql function associated with i-th SQL parameter*.

b) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by

Case:

i) If TSF_i is an SQL-invoked method, then the <parameter type> of the second SQL parameter of TSF_i.

ii) Otherwise, the <parameter type> of the first SQL parameter of TSF_i.

3) Otherwise:

a) The Syntax Rules of Subclause 9.19, "Determination of a from-sql function", are applied with the data type identified by T_i and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable from-sql function FSF_i. FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

b) The Syntax Rules of Subclause 9.21, "Determination of a to-sql function", are applied with the data type identified by T_i and the <group name>

contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function TSF_i. TSF_i is called the *to-sql function associated with the i-th SQL parameter*.

c) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

C) Otherwise, the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration>.

2) Effective SQL parameter list entries *PN*+1 to *PN*+*N* are *N* occurrences of an SQL parameter of an implementation-defined <data type> that is an exact numeric type with scale 0. The <parameter mode> for the *i*-th such effective SQL parameter is the same as that of the *i*–*PN*-th effective SQL parameter.

3) Effective SQL parameter list entry (*PN*+*N*)+1 is an SQL parameter of a <data type> that is character string of length 5 and character set SQL_TEXT with <parameter mode> INOUT.

4) Effective SQL parameter list entry (*PN*+*N*)+2 is an SQL parameter of a <data type> that is character string of implementation-defined length and character set SQL_TEXT with <parameter mode> IN.

5) Effective SQL parameter list entry (*PN*+*N*)+3 is an SQL parameter of a <data type> that is character string of implementation-defined length and character set SQL_TEXT with <parameter mode> IN.

6) Effective SQL parameter list entry (*PN*+*N*)+4 is an SQL parameter of a <data type> that is character string of implementation-defined length and character set SQL_TEXT with <parameter mode> INOUT.

e) If PARAMETER STYLE GENERAL is specified, then let the *effective SQL parameter list* be a list of *PN* parameters such that, for *i* ranging from 1 (one) to *PN*, the *i*-th effective SQL parameter list entry is defined as follows.

Case:

i) If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> simply contains <locator indication>, then the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by INTEGER.

ii) If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> is a <path-resolved user-defined type name> without a <locator indication>, then:

1) Case:

A) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is IN, then:

I) The Syntax Rules of Subclause 9.19, “Determination of a from-sql function”, are applied with the data type identified by T_i and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable from-sql function FSF_i. FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

II) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

B) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is OUT, then:

I) The Syntax Rules of Subclause 9.21, "Determination of a to-sql function", are applied with the data type identified by T_i and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function TSF_i. TSF_i is called the *to-sql function associated with the i-th SQL parameter*.

II) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by

Case:

1) If TSF_i is an SQL-invoked method, then the <parameter type> of the second SQL parameter of TSF_i.

2) Otherwise, the <parameter type> of the first SQL parameter of TSF_i.

C) Otherwise:

I) The Syntax Rules of Subclause 9.19, "Determination of a from-sql function", are applied with the data type identified by T_i and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable from-sql function FSF_i. FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

II) The Syntax Rules of Subclause 9.21, "Determination of a to-sql function", are applied with the data type identified by T_i and the <group name> contained in the <group specification> that contains T_i as *TYPE* and *GROUP*, respectively. There shall be an applicable to-sql function TSF_i. TSF_i is called the *to-sql function associated with the i-th SQL parameter*.

III) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

iii) Otherwise, the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration>.

NOTE 339 — If the SQL-invoked routine is an SQL-invoked function, then the value returned from the external routine is passed to the SQL-implementation in an implementation-dependent manner. An SQL parameter is not used for this purpose.

f) Depending on whether the <language clause> specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the *operative data type correspondences table* be Table 16, "Data type correspondences for Ada", Table 17, "Data type correspondences for C", Table 18, "Data type correspondences for COBOL", Table 19, "Data type correspondences for Fortran", Table 20, "Data type correspondences for M", Table 21, "Data type correspondences for Pascal", or Table 22, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the "SQL data type" column and the "host data type column".

g) Any <data type> in an effective SQL parameter list entry shall specify a data type listed in the SQL data type column for which the corresponding row in the host data type column is not "None".

21) Case:

a) If <method specification designator> is specified, then:

i) *R* is deterministic if *DMS* indicates that the method is deterministic; otherwise, *R* is possibly non-deterministic.

ii) *R* possibly modifies SQL-data if the SQL-data access indication of *DMS* indicates that the method possibly modifies SQL-data. *R* possibly reads SQL-data if the SQL-data access indication of *DMS* indicates that the method possibly reads SQL-data. *R* possibly contains SQL if the SQL-data access indication of *DMS* indicates that the method possibly contains SQL. Otherwise, *R* does not possibly contain SQL.

b) Otherwise:

i) If DETERMINISTIC is specified, then *R* is *deterministic*; otherwise, it is *possibly non-deterministic*.

ii) An <SQL-invoked routine> *possibly modifies SQL-data* if and only if <SQL-data access indication> specifies MODIFIES SQL DATA.

iii) An <SQL-invoked routine> *possibly reads SQL-data* if and only if <SQL-data access indication> specifies READS SQL DATA.

iv) An <SQL-invoked routine> *possibly contains SQL* if and only if <SQL-data access indication> specifies CONTAINS SQL.

v) An <SQL-invoked routine> *does not possibly contain SQL* if and only if <SQL-data access indication> specifies NO SQL.

22) If *R* is a schema-level routine, then let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <schema qualified routine name>.

23) If the <SQL-invoked routine> is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>; otherwise, let *A* be the <authorization identifier> that owns the schema identified by the explicit or implicit <schema name> of the <schema qualified routine name>.

Access Rules

1) If an <SQL-invoked routine> is contained in an <SQL-client module definition> *M* with no intervening <schema definition>, then the enabled authorization identifiers shall include the <authorization identifier> that owns *S*.

2) If *R* is an external routine and if any of its SQL parameters have an associated from-sql function or a to-sql function, or if *R* has a to-sql function associated with the result, then

Case:

a) If <SQL-invoked routine> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include EXECUTE on all from-sql functions (if any) and on all to-sql functions (if any) associated with the SQL parameters and on the to-sql function associated with the result (if any).

b) Otherwise, the current privileges shall include EXECUTE on all from-sql functions (if any) and on all to-sql functions (if any) associated with the SQL parameters and on the to-sql function associated with the result (if any).

General Rules

1) If *R* is a schema-level routine, then a privilege descriptor is created that defines the EXECUTE privilege on *R* to the <authorization identifier> that owns the schema that includes *R*. The grantor for the privilege descriptor is set to the special grantor value "_SYSTEM". This privilege is grantable if and only if one of the following is satisfied:

   a) *R* is an SQL routine and all of the privileges necessary for the <authorization identifier> to successfully execute the <SQL procedure statement> contained in the <routine body> are grantable. The necessary privileges include the EXECUTE privilege on every subject routine of every <routine invocation> contained in the <SQL procedure statement>.

   b) *R* is an SQL routine and SQL SECURITY INVOKER is specified.

   c) *R* is an external routine.

2) Case:

   a) If <SQL-invoked routine> is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

   b) If <SQL-invoked routine> is contained in a <preparable statement> or in a <direct SQL statement>, then let *DP* be the SQL-path of the current SQL-session.

   c) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains <SQL-invoked routine>.

3) If <method specification designator> is not specified, then a routine descriptor is created that describes the SQL-invoked routine being defined:

   a) The routine name included in the routine descriptor is <schema qualified routine name>.

   b) The specific name included in the routine descriptor is <specific name>.

   c) The routine descriptor includes, for each SQL parameter in *NPL*, the name, declared type, ordinal position, an indication of whether the SQL parameter is input, output, or both, and an indication of whether the SQL parameter is a RESULT SQL parameter.

   d) The maximum number of returned result sets included in the routine descriptor is

      Case:

      i) If the SQL-invoked routine is an SQL-invoked procedure, then the explicit or implicit value of <maximum returned result sets>.

      ii) Otherwise, 0 (zero).

   e) The routine descriptor includes an indication of whether the SQL-invoked routine is an SQL-invoked function or an SQL-invoked procedure.

   f) If the SQL-invoked routine is an SQL-invoked function, then:

i) The routine descriptor includes an indication that the SQL-invoked function is not an SQL-invoked method.

ii) The routine descriptor includes the data type in the <returns data type>. If the <returns data type> simply contains <locator indication>, then the routine descriptor includes an indication that the return value is a locator.

iii) The SQL-invoked routine descriptor includes an indication of whether the SQL-invoked routine is a null-call function.

g) If the SQL-invoked routine is a type-preserving function, then the routine descriptor includes an indication that the SQL-invoked routine is a type-preserving function.

h) The name of the language in which the body of the SQL-invoked routine was written is the <language name> contained in the <language clause>.

i) If the SQL-invoked routine is an SQL routine, then the SQL routine body of the routine descriptor is the <SQL routine body>.

j) If the SQL-invoked routine is an SQL-invoked function or NEW SAVEPOINT LEVEL is specified, then the routine descriptor includes an indication that a new savepoint level is to be established whenever the routine is invoked.

NOTE 340 — The use of savepoint levels is dependent on Feature T272, “Enhanced savepoint management”.

k) Case:

i) If SQL SECURITY INVOKER is specified, then the SQL security characteristic in the routine descriptor is INVOKER.

ii) Otherwise, the SQL security characteristic in the routine descriptor is DEFINER.

l) If the SQL-invoked routine is an external routine, then:

i) The external name of the routine descriptor is <external routine name>.

ii) The routine descriptor includes an indication of whether the *parameter passing style* is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

m) The SQL-invoked routine descriptor includes an indication of whether the SQL-invoked routine is DETERMINISTIC or NOT DETERMINISTIC.

n) The routine descriptor includes an indication of whether the SQL-invoked routine does not possibly contain SQL, possibly contains SQL, possibly reads SQL-data, or possibly modifies SQL-data.

o) If the SQL-invoked routine specifies a <result cast>, then the routine descriptor includes an indication that the SQL-invoked routine specifies a <result cast> and the <data type> specified in the <result cast>. If <result cast> contains <locator indication>, then the routine descriptor includes an indication that the <data type> specified in the <result cast> has a locator indication.

p) For every SQL parameter that has an associated from-sql function *FSF*, the routine descriptor includes the specific name of *FSF*.

q) For every SQL parameter that has an associated to-sql function *TSF*, the routine descriptor includes the specific name of *TSF*.

r) If *R* is an external function and if *R* has a to-sql function associated with its result *TRF*, then the routine descriptor includes the specific name of *TRF*.

s) For every SQL parameter whose <SQL parameter declaration> contains <locator indication>, the routine descriptor includes an indication that the SQL parameter is a locator parameter.

t) The routine authorization identifier included in the routine descriptor is the <authorization identifier> that owns *S*.

u) The routine SQL-path included in the routine descriptor is *DP*.

NOTE 341 — The routine SQL-path is used to set the routine SQL-path of the current SQL-session when *R* is invoked. The routine SQL-path of the current SQL-session is used by the Syntax Rules of Subclause 10.4, "<routine invocation>", to define the subject routines of <routine invocation>s contained in *R*. The same routine SQL-path is used whenever *R* is invoked.

v) The routine descriptor includes an indication that the routine is a schema-level routine.

w) The routine descriptor includes an indication of whether the SQL-invoked routine is dependent on a user-defined type.

NOTE 342 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.27, "SQL-invoked routines".

4) If <method specification designator> is specified, then let *DMS* be the descriptor of the corresponding method specification. A routine descriptor is created that describes the SQL-invoked routine being defined.

a) The routine name included in the routine descriptor is *RN*.

b) The specific name included in the routine descriptor is <specific name>.

c) The routine descriptor includes, for each SQL parameter in *NPL*, the name, data type, ordinal position, an indication of whether the SQL parameter is input, output, or both, and an indication of whether the SQL parameter is a RESULT SQL parameter.

d) The routine descriptor includes an indication that the SQL-invoked routine is an SQL-invoked function that is an SQL-invoked method, an indication of the user-defined type *UDT*, and an indication of whether STATIC or CONSTRUCTOR was specified.

e) If the SQL-invoked routine is a type-preserving function, then the routine descriptor includes an indication that the SQL-invoked routine is a type-preserving function.

f) If the SQL-invoked routine is a mutator function, then the routine descriptor includes an indication that the SQL-invoked routine is a mutator function.

g) The routine descriptor includes the data type in the <returns data type>.

h) The name of the language in which the body of the SQL-invoked routine was written is the <language name> contained in the <language clause> in *DMS*.

i) If the SQL-invoked routine is an SQL routine, then the SQL routine body of the routine descriptor is the <SQL routine body>.

j) Case:

i) If SQL SECURITY INVOKER is specified, then the SQL security characteristic in the routine descriptor is INVOKER.

ii) Otherwise, the SQL security characteristic in the routine descriptor is DEFINER.

k) If the SQL-invoked routine is an external routine, then:

i) The external name of the routine descriptor is <external routine name>.

ii) The routine descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL, which is the same as the indication of <parameter style> in *DMS*.

l) The routine descriptor includes an indication of whether the SQL-invoked routine is deterministic.

m) The routine descriptor includes an indication of whether the SQL-invoked routine possibly modifies SQL-data, possibly read SQL-data, possibly contains SQL, or does not possibly contain SQL.

n) The routine descriptor includes an indication of whether the SQL-invoked routine is a null-call function, which is the same as the indication in *DMS*.

o) If *DMS* specifies a <result cast>, then the routine descriptor includes an indication that the SQL-invoked routine specifies a <result cast> and the <data type> specified in the <result cast> of *DMS*.

p) The routine authorization identifier included in the routine descriptor is the <authorization identifier> that owns *S*.

q) The routine SQL-path included in the routine descriptor is *DP*.

NOTE 343 — The routine SQL-path is used to set the routine SQL-path of the current SQL-session when *R* is invoked. The routine SQL-path of the current SQL-session is used by the Syntax Rules of Subclause 10.4, "<routine invocation>", to define the subject routine of <routine invocation>s contained in *R*. The same routine SQL-path is used whenever *R* is invoked.

r) The routine descriptor includes an indication of whether the routine is a schema-level routine.

s) The routine descriptor includes an indication of whether the SQL-invoked routine is dependent on a user-defined type.

NOTE 344 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.27, "SQL-invoked routines".

5) The creation timestamp and the last-altered timestamp included in the routine descriptor are the values of CURRENT_TIMESTAMP.

6) If *R* is an external routine, then the routine descriptor of *R* includes further elements determined as follows:

a) Case:

i) If <SQL-data access indication> in the descriptor of *R* is MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL, then:

1) Let *P* be the program identified by the <external routine name>.

2) The external routine SQL-path is the <schema name list> immediately contained in the <path specification> that is immediately contained in the <module path specification> of the <SQL-client module definition> of *P*.

ii) Otherwise, the external routine SQL-path is implementation-defined.

NOTE 345 — The external routine SQL-path is used to set the routine SQL-path of the current SQL-session when *R* is invoked. The routine SQL-path of the current SQL-session is used by the Syntax Rules of Subclause 10.4, "<routine invocation>", to define the subject routines of <routine invocation>s contained in the <SQL-client module definition> of *P*. The same external routine SQL-path is used whenever *R* is invoked.

b) The external security characteristic in the routine descriptor is

Case:

i) If <external security clause> specifies EXTERNAL SECURITY DEFINER, then DEFINER.

ii) If <external security clause> specifies EXTERNAL SECURITY INVOKER, then INVOKER.

iii) Otherwise, EXTERNAL SECURITY IMPLEMENTATION DEFINED.

c) The effective SQL parameter list is the *effective SQL parameter list*.

Conformance Rules

1) Without Feature T471, “Result sets return value”, conforming SQL language shall not contain a <returned result sets characteristic>.

2) Without Feature T322, “Overloading of SQL-invoked functions and procedures”, conforming SQL language shall not contain a <schema routine> in which the schema identified by the explicit or implicit schema name of the <schema qualified routine name> includes a routine descriptor whose routine name is <schema qualified routine name>.

3) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <method specification designator>.

4) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <transform group specification>.

5) Without Feature S024, “Enhanced structured types”, an <SQL parameter declaration> shall not contain RESULT.

6) Without Feature T571, “Array-returning external SQL-invoked functions”, conforming SQL language shall not contain an <SQL-invoked routine> that defines an array-returning external function.

7) Without Feature T572, “Multiset-returning external SQL-invoked functions”, conforming SQL language shall not contain an <SQL-invoked routine> that defines a multiset-returning external function.

8) Without Feature S201, “SQL-invoked routines on arrays”, conforming SQL language shall not contain a <parameter type> that is based on an array type.

9) Without Feature S201, “SQL-invoked routines on arrays”, conforming SQL language shall not contain a <returns data type> that is based on an array type.

10) Without Feature S202, “SQL-invoked routines on multisets”, conforming SQL language shall not contain a <parameter type> that is based on a multiset type.

11) Without Feature S202, “SQL-invoked routines on multisets”, conforming SQL language shall not contain a <returns data type> that is based on a multiset type.

12) Without Feature T323, “Explicit security for external routines”, conforming SQL language shall not contain an <external security clause>.

13) Without Feature S231, “Structured type locators”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a structured type.

14) Without Feature S231, “Structured type locators”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a structured type.

15) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies an array type.

16) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies an array type.

17) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a multiset type.

18) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a multiset type.

19) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a large object type.

20) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a large object type.

21) Without Feature S027, “Create method by specific method name”, conforming SQL language shall not contain a <method specification designator> that contains SPECIFIC METHOD.

22) Without Feature T324, “Explicit security for SQL routines”, conforming SQL language shall not contain a <rights clause>.

23) Without Feature T326, “Table functions”, conforming SQL language shall not contain a <returns table type>.

24) Without Feature T651, “SQL-schema statements in SQL routines”, conforming SQL language shall not contain an <SQL routine body> that contains an SQL-schema statement.

25) Without Feature T652, “SQL-dynamic statements in SQL routines”, conforming SQL language shall not contain an <SQL routine body> that contains an SQL-dynamic statement.

26) Without Feature T653, “SQL-schema statements in external routines”, conforming SQL language shall not contain an <external routine name> that identifies a program in which an SQL-schema statement appears.

27) Without Feature T654, “SQL-dynamic statements in external routines”, conforming SQL language shall not contain an <external routine name> that identifies a program in which an SQL-dynamic statement appears.

28) Without Feature T655, “Cyclically dependent routines”, conforming SQL language shall not contain an <SQL routine body> that contains a <routine invocation> whose subject routine is generally dependent on the routine descriptor of the SQL-invoked routine specified by <SQL-invoked routine>.

29) Without Feature T272, “Enhanced savepoint management”, conforming SQL language shall not contain a <routine characteristics> that contains a <savepoint level indication>.

30) Without Feature B121, “Routine language Ada”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains ADA.

31) Without Feature B122, “Routine language C”, conforming SQL language shall not contain a <routine characteristic>that contains a <language clause> that contains C.

32) Without Feature B123, “Routine language COBOL”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains COBOL.

33) Without Feature B124, “Routine language Fortran”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains FORTRAN.

34) Without Feature B125, “Routine language MUMPS”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains M.

35) Without Feature B126, “Routine language Pascal”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains PASCAL.

36) Without Feature B127, “Routine language PL/I”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains PLI.

37) Without Feature B128, “Routine language SQL”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains SQL.

11.51 <alter routine statement>

This Subclause is modified by Subclause 9.9, "<alter routine statement>", in ISO/IEC 9075-13.

Function

Alter a characteristic of an SQL-invoked routine.

Format

```
<alter routine statement> ::=
  ALTER <specific routine designator>
      <alter routine characteristics> <alter routine behavior>

<alter routine characteristics> ::=
  <alter routine characteristic>...

<alter routine characteristic> ::=
    <language clause>
  | <parameter style clause>
  | <SQL-data access indication>
  | <null-call clause>
  | <returned result sets characteristic>
  | NAME <external routine name>

<alter routine behavior> ::=
  RESTRICT
```

Syntax Rules

1) Let *SR* be the SQL-invoked routine identified by the <specific routine designator> and let *SN* be the <specific name> of *SR*. The schema identified by the explicit or implicit <schema name> of *SN* shall include the descriptor of *SR*.

2) *SR* shall be a schema-level routine.

3) *SR* shall not be an SQL-invoked routine that is dependent on a user-defined type.

   NOTE 346 — "SQL-invoked routine dependent on a user-defined type" is defined in Subclause 4.27, "SQL-invoked routines".

4) If RESTRICT is specified, then:

   a) *SR* shall not be the ordering function included in the descriptor of any user-defined type *UDT*.

   b) *SR* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The <query expression> of any view descriptor.

      iii) The <search condition> of any constraint descriptor.

      iv) The triggered action of any trigger descriptor.

c) *SN* shall not be included in any of the following:

i) A group descriptor of any transform descriptor.

ii) A user-defined cast descriptor.

5) *SR* shall be an external routine.

6) *SR* shall not be an SQL-invoked method that is an overriding method and the set of overriding methods of *SR* shall be empty.

7) <alter routine characteristics> shall contain at most one <language clause>, at most one <parameter style clause>, at most one <SQL-data access indication>, at most one <null-call clause>, at most one <returned result sets characteristic>, and at most one <external routine name>.

8) If <returned result sets characteristic> is specified, then *SR* shall be an SQL-invoked procedure.

9) If <language clause> is specified, then:

a) <language clause> shall not specify SQL.

b) Depending on whether the <language clause> specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the operative data type correspondences table be Table 16, "Data type correspondences for Ada", Table 17, "Data type correspondences for C", Table 18, "Data type correspondences for COBOL", Table 19, "Data type correspondences for Fortran", Table 20, "Data type correspondences for M", Table 21, "Data type correspondences for Pascal", or Table 22, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the "SQL data type" column and the "host data type column".

c) Any <data type> in the effective SQL parameter list entry of *SR* shall specify a data type listed in the SQL data type column for which the corresponding row in the host data type column is not "None".

NOTE 347 — "Effective SQL parameter list" is defined in Subclause 11.50, "<SQL-invoked routine>".

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *SN*.

General Rules

1) If *SR* is not a method, then the routine descriptor of *SR* is modified:

a) If <returned result sets characteristic> is specified, then the value of <maximum returned result sets>.

b) If <language clause> is specified, then the <language name> contained in the <language clause>.

c) If <external routine name> is specified, then the external name of the routine descriptor is <external routine name>.

d) If <parameter style clause> is specified, then the routine descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

e) If the <SQL-data access indication> is specified, then an indication of whether the SQL-invoked routine's <SQL-data access indication> is READS SQL DATA, MODIFIES SQL DATA, CONTAINS SQL, or NO SQL.

f) If <null-call clause> is specified, then an indication of whether the SQL-invoked routine is a null-call function.

2) If *SR* is a method, then let *DMS* be the descriptor of the corresponding method specification. *DMS* is modified:

a) If <language clause> is specified, then the <language name> contained in the <language clause>.

b) If <parameter style clause> is specified, then the method specification descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

c) If the <SQL-data access indication> is specified, then an indication of whether the SQL-invoked routine's <SQL-data access indication> is READS SQL DATA, MODIFIES SQL DATA, CONTAINS SQL, or NO SQL.

d) If <null-call clause> is specified, then an indication of whether the method should not be invoked if any argument is the null value.

3) If *SR* is a method, then the routine descriptor of *SR* is modified:

a) If <external routine name> is specified, then the external name of the routine descriptor is <external routine name>. If <parameter style clause> is specified, then the method specification descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

Conformance Rules

1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <alter routine statement>.

11.52 <drop routine statement>

This Subclause is modified by Subclause 9.25, "<drop routine statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.11, "<drop routine statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 9.10, "<drop routine statement>", in ISO/IEC 9075-13.

Function

Destroy an SQL-invoked routine.

Format

```
<drop routine statement> ::=
  DROP <specific routine designator> <drop behavior>
```

Syntax Rules

1) Let *SR* be the SQL-invoked routine identified by the <specific routine designator> and let *SN* be the <specific name> of *SR*. The schema identified by the explicit or implicit <schema name> of *SN* shall include the descriptor of *SR*.

2) *SR* shall be a schema-level routine.

3) *SR* shall not be dependent on any user-defined type.

   NOTE 348 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.27, "SQL-invoked routines".

4) If RESTRICT is specified, then *SR* shall not be the ordering function included in the descriptor of any user-defined type *UDT*.

5) If RESTRICT is specified, then:

   a) *SR* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The <query expression> of any view descriptor.

      iii) The <search condition> of any constraint descriptor.

      iv) The triggered action of any trigger descriptor.

   b) *SN* shall not be a included in any of the following:

      i) A group descriptor of any transform descriptor.

      ii) A user-defined cast descriptor.

   NOTE 349 — If CASCADE is specified, then such referencing objects will be dropped by the execution of the <revoke statement> specified in the General Rules of this Subclause.

6) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in *SN*.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of *SN*. The enabled authorization identifiers shall include *A*.

General Rules

1) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE EXECUTE ON SPECIFIC ROUTINE SN FROM A CASCADE
```

2) Let *DN* be the <user-defined type name> of a user-defined type whose descriptor includes *SN* in the group descriptor of any transform descriptor. Let *GN* be the <group name> of that group descriptor. The following <drop transform statement> is effectively executed without further Access Rule checking:

```
DROP TRANSFORM GN FOR DN CASCADE
```

3) Let *UDCD* be a user-defined cast descriptor that includes *SN* as its cast function. Let *SDT* be the source data type included in *UDCD*. Let *TDT* be the target data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( DN AS TD ) CASCADE
```

4) If *SR* is the ordering function included in the descriptor of a user-defined type *UDT*, then let *UDTN* be a <path-resolved user-defined type name> that identifies *UDT*. The following <drop user-defined ordering statement> is effectively executed without further Access Rule checking:

```
DROP ORDERING FOR UDTN CASCADE
```

5) The descriptor of *SR* is destroyed.

Conformance Rules

1) Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop routine statement> that contains a <drop behavior> that contains CASCADE.

2) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <drop routine statement> that contains a <specific routine designator> that identifies a method.

11.53 <user-defined cast definition>

This Subclause is modified by Subclause 11.12, "<user-defined cast definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 12.8, "<user-defined cast definition>", in ISO/IEC 9075-14.

Function

Define a user-defined cast.

Format

```
<user-defined cast definition> ::=
  CREATE CAST <left paren> <source data type> AS <target data type> <right paren>
      WITH <cast function>
      [ AS ASSIGNMENT ]

<cast function> ::=
  <specific routine designator>

<source data type> ::=
  <data type>

<target data type> ::=
  <data type>
```

Syntax Rules

1) Let *SDT* be the <source data type>. The data type identified by *SDT* is called the *source data type*.

2) Let *TDT* be the <target data type>. The data type identified by *TDT* is called the *target data type*.

3) There shall be no user-defined cast for *SDT* and *TDT*.

4) At least one of *SDT* or *TDT* shall contain a <schema-resolved user-defined type name> or a <reference type>.

5) If *SDT* contains a <schema-resolved user-defined type name>, then let *SSDT* be the schema that includes the descriptor of the user-defined type identified by *SDT*.

6) If *SDT* contains a <reference type>, then let *SSDT* be the schema that includes the descriptor of the referenced type of the reference type identified by *SDT*.

7) If *TDT* contains a <schema-resolved user-defined type name>, then let *STDT* be the schema that includes the descriptor of the user-defined type identified by *TDT*.

8) If *TDT* contains a <reference type>, then let *STDT* be the schema that includes the descriptor of the referenced type of the reference type identified by *TDT*.

9) If both *SDT* and *TDT* contain a <schema-resolved user-defined type name> or a <reference type>, then the <authorization identifier> that owns *SSDT* and the <authorization identifier> that owns *STDT* shall be equivalent.

10) Let *F* be the SQL-invoked routine identified by <cast function>. *F* is called the *cast function* for source data type *SDT* and target data type *TDT*.

    a) *F* shall have exactly one SQL parameter, and its declared type shall be *SDT*.

    b) The result data type of *F* shall be *TDT*.

    c) The <authorization identifier> that owns *SSDT* or *STDT* (both, if both *SDT* and *TDT* are <schema-resolved user-defined type name>s) shall own the schema that includes the SQL-invoked routine descriptor of *F*.

    d) *F* shall be deterministic.

    e) *F* shall not possibly modify SQL-data.

    f) *F* shall not possibly read SQL-data.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema that includes the routine descriptor of *F*.

2) If *SDT* contains a <schema-resolved user-defined type name> or a <reference type>, then the enabled authorization identifiers shall include the <authorization identifier> that owns *SSDT*.

3) If *TDT* contains a <schema-resolved user-defined type name> or a <reference type>, then the enabled authorization identifiers shall include the <authorization identifier> that owns *STDT*.

General Rules

1) A user-defined cast descriptor *CFD* is created that describes the user-defined cast. *CFD* includes the name of the source data type, the name of the target data type, the specific name of the cast function, and, if and only if AS ASSIGNMENT is specified, an indication that the cast function is implicitly invocable.

Conformance Rules

1) Without Feature S211, “User-defined cast functions”, conforming SQL language shall not contain a <user-defined cast definition>.

11.54 <drop user-defined cast statement>

This Subclause is modified by Subclause 9.26, "<drop user-defined cast statement>", in ISO/IEC 9075-4.

Function

Destroy a user-defined cast.

Format

```
<drop user-defined cast statement> ::=
  DROP CAST <left paren> <source data type> AS <target data type> <right paren>
      <drop behavior>
```

Syntax Rules

1) Let *SDT* be the <source data type> and let *TDT* be the <target data type>.

2) Let *CF* be the user-defined cast whose user-defined cast descriptor includes *SDT* as the source data type and *TDT* as the target data type.

3) Let *SN* be the specific name of the cast function *F* included in the user-defined cast descriptor of *CF*.

4) The schema identified by the <schema name> of *SN* shall include the descriptor of *F*.

5) Let *CS* be any <cast specification> such that:

   a) The <value expression> of *CS* has declared type *P*.

   b) The <cast target> of *CS* is either *TDT* or a domain with declared type *TDT*.

   c) The type designator of *SDT* is in the type precedence of *P*.

   d) No other data type *Q* whose type designator precedes *SDT* in the type precedence list of *P* such that there is a user-defined cast CF_q whose user-defined cast descriptor includes *Q* as the source data type and *TDT* as the target data type.

6) Let *PS* be any SQL procedure statement that is dependent on *F*.

7) If RESTRICT is specified, then neither *CS* nor *PS* shall be generally contained in any of the following:

   a) The SQL routine body of any routine descriptor.

   b) The <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) The trigger action of any trigger descriptor.

   NOTE 350 — If CASCADE is specified, then such referencing objects will be dropped as specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifier shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *SN*.

General Rules

1) Let *R* be any SQL-invoked routine that contains *CS* or *PS* in its SQL routine body. Let *SN* be the specific name of *R*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

2) Let *V* be any view that contains *CS* or *PS* in its <query expression>. Let *VN* be the <table name> of *V*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW VN CASCADE
```

3) Let *T* be any table that contains *CS* or *PS* in the <search condition> of any constraint descriptor included in the table descriptor of *T*. Let *TN* be the <table name> of *T*. The following <drop table statement> is effectively executed without further Access Rule checking:

```
DROP TABLE TN CASCADE
```

4) Let *A* be any assertion that contains *CS* or *PS* in its <search condition>. Let *AN* be the <constraint name> of *A*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION AN CASCADE
```

5) Let *D* be any domain that contains *CS* or *PS* in the <search condition> of any constraint descriptor. Let *DN* be the <domain name> of *D*. The following <drop domain statement> is effectively executed without further Access Rule checking:

```
DROP DOMAIN DN CASCADE
```

6) Let *T* be any trigger whose trigger descriptor includes a trigger action that contains *CS* or *PS*. Let *TN* be the <trigger name> of *T*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER TN
```

7) The descriptor of *CF* is destroyed.

Conformance Rules

1) Without Feature S211, “User-defined cast functions”, conforming SQL language shall not contain a <drop user-defined cast statement>.

11.55 <user-defined ordering definition>

This Subclause is modified by Subclause 11.13, "<user-defined ordering definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 9.11, "<user-defined ordering definition>", in ISO/IEC 9075-13.

Function

Define a user-defined ordering for a user-defined type.

Format

```
<user-defined ordering definition> ::=
  CREATE ORDERING FOR <schema-resolved user-defined type name> <ordering form>

<ordering form> ::=
    <equals ordering form>
  | <full ordering form>

<equals ordering form> ::=
  EQUALS ONLY BY <ordering category>

<full ordering form> ::=
  ORDER FULL BY <ordering category>

<ordering category> ::=
    <relative category>
  | <map category>
  | <state category>

<relative category> ::=
  RELATIVE WITH <relative function specification>

<map category> ::=
  MAP WITH <map function specification>

<state category> ::=
  STATE [ <specific name> ]

<relative function specification> ::=
  <specific routine designator>

<map function specification> ::=
  <specific routine designator>
```

Syntax Rules

1) Let *UDTN* be the <schema-resolved user-defined type name>. Let *UDT* be the user-defined type identified by *UDTN*.

2) The descriptor of *UDT* shall include an ordering form that specifies NONE.

3) If *UDT* is not a maximal supertype, then

   Case:

a) If <equals ordering form> is specified, then the comparison form of every direct supertype of *UDT* shall be EQUALS.

b) Otherwise, the comparison form of every direct supertype of *UDT* shall be FULL.

4) If <relative category> or <state category> is specified, then *UDT* shall be a maximal supertype.

5) If <map category> is specified and *UDT* is not a maximal supertype, then the comparison category of every direct supertype of *UDT* shall be MAP.

NOTE 351 — The comparison categories of two user-defined types in the same subtype family shall be the same.

6) Case:

a) If <state category> is specified, then

i) *UDT* shall not be a distinct type.

ii) EQUALS ONLY shall be specified.

iii) The declared type of each attribute of *UDT* shall not be UDT-NC-ordered.

iv) Case:

1) If <specific name> is specified, then let *SN* be <specific name>. If *SN* contains a <schema name>, then that <schema name> shall be equivalent to the <schema name> of *UDTN*.

2) Otherwise, let *SN* be an implementation-dependent <specific name> whose <schema name> is equivalent to the <schema name> *S* of *UDTN*. This implementation-dependent <specific name> shall not be equivalent to the <specific name> of any other routine descriptor in the schema identified by *S*.

b) Otherwise:

i) Let *F* be the SQL-invoked routine identified by the <specific routine designator> *SRD*.

ii) *F* shall be deterministic.

iii) *F* shall not possibly modify SQL-data.

7) If <relative function specification> is specified, then:

a) *F* shall have exactly two SQL parameters whose declared type is *UDT*.

b) *F* shall be an SQL-invoked regular function.

c) The result data type of *F* shall be INTEGER.

8) If <map function specification> is specified, then:

a) *F* shall have exactly one SQL parameter whose declared type is *UDT*.

b) The result data type of *F* shall be a predefined data type.

c) The result data type of *F* is an operand of an equality operation. The Syntax Rules of Subclause 9.9, “Equality operations”, apply.

d) If FULL is specified, then the result data type of *F* is an operand of an ordering operation. The Syntax Rules of Subclause 9.12, “Ordering operations”, apply.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema that includes the descriptor of *UDT* and the schema that includes the routine descriptor of *F*.

General Rules

1) If <state category> is specified, then:

   a) Let C_1, ..., C_n be the components of the representation of the user-defined type.

   b) Let *SNUDT* be the <schema name> of the schema that includes the descriptor of *UDT*.

   c) The following <SQL-invoked routine> is effectively executed:

```
CREATE FUNCTION SNUDT.EQUALS ( UDT1 UDTN, UDT2 UDTN )
    RETURNS BOOLEAN
    SPECIFIC SN
    DETERMINISTIC
    CONTAINS SQL
    STATIC DISPATCH
    RETURN
      ( TRUE AND
        UDT1.SPECIFICTYPE = UDT2.SPECIFICTYPE AND
        UDT1.C1 = UDT2.C1 AND
        ...
        UDT1.Cn = UDT2.Cn )
```

2) Case:

   a) If EQUALS is specified, then the ordering form in the user-defined type descriptor of *UDT* is set to EQUALS.

   b) Otherwise, the ordering form in the user-defined type descriptor of *UDT* is set to FULL.

3) Case:

   a) If RELATIVE is specified, then the ordering category in the user-defined type descriptor of *UDT* is set to RELATIVE.

   b) If MAP is specified, then the ordering category in the user-defined type descriptor of *UDT* is set to map.

   c) Otherwise, the ordering category in the user-defined type descriptor of *UDT* is set to STATE.

4) The <specific routine designator> identifying the ordering function, depending on the ordering category, in the descriptor of *UDT* is set to *SRD*.

Conformance Rules

1) Without Feature S251, “User-defined orderings”, conforming SQL shall not contain a <user-defined ordering definition>.

NOTE 352 — If MAP is specified, then the Conformance Rules of Subclause 9.9, “Equality operations”, apply. If ORDER FULL BY MAP is specified, then the Conformance Rules of Subclause 9.12, “Ordering operations”, also apply.

11.56 <drop user-defined ordering statement>

This Subclause is modified by Subclause 9.20, "<drop user-defined ordering statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 9.12, "<drop user-defined ordering statement>", in ISO/IEC 9075-13.

Function

Destroy a user-defined ordering method.

Format

```
<drop user-defined ordering statement> ::=
  DROP ORDERING FOR <schema-resolved user-defined type name> <drop behavior>
```

Syntax Rules

1) Let *UDTN* be the <schema-resolved user-defined type name>. Let *UDT* be the user-defined type identified by *UDTN*.

2) The descriptor of *UDT* shall include an ordering form that specifies EQUALS or FULL.

3) Let *OF* be the ordering function of *UDT*.

4) If RESTRICT is specified, then none of the following shall contain an operand of an equality operation, grouping operation or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*:

   a) The SQL routine body of any routine descriptor.

   b) The <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) The triggered action of any trigger descriptor.

   NOTE 353 — If CASCADE is specified, then such referencing objects will be dropped as specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *UDTN*.

General Rules

1) Let *R* be any SQL-invoked routine whose SQL routine body contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*. Let *SN* be the specific name of *R*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

2) Let *V* be any view whose <query expression> contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*. Let *VN* be the <table name> of *V*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW VN CASCADE
```

3) Let *T* be any table whose table descriptor includes a constraint descriptor of a constraint *C* whose <search condition> contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*. Let *TN* be the <table name> of *T* . Let *TCN* be the <constraint name> of *C* . The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE TN DROP CONSTRAINT TCN CASCADE
```

4) Let *A* be any assertion whose <search condition> contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*. Let *AN* be the <constraint name> of *A*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION AN CASCADE
```

5) Let *D* be any domain whose descriptor includes a constraint descriptor that includes an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT* in the <search condition> of any constraint descriptor or in the <default option> included in the domain descriptor of *D*. Let *DN* be the <domain name> of *D* . The following <drop domain statement> is effectively executed without further Access Rule checking:

```
DROP DOMAIN DN CASCADE
```

6) Let *T* be any trigger whose triggered action contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*. Let *TN* be the <trigger name> of *T*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER TN
```

7) In the descriptor of *UDT*, the ordering form is set to NONE and the ordering category is set to STATE. No ordering function is included in the descriptor of *UDT*.

Conformance Rules

1) Without Feature S251, “User-defined orderings”, conforming SQL language shall not contain a <drop user-defined ordering statement>.

11.57 <transform definition>

Function

Define one or more transform functions for a user-defined type.

Format

```
<transform definition> ::=
  CREATE { TRANSFORM | TRANSFORMS } FOR
      <schema-resolved user-defined type name> <transform group>...

<transform group> ::=
  <group name> <left paren> <transform element list> <right paren>

<group name> ::=
  <identifier>

<transform element list> ::=
  <transform element> [ <comma> <transform element> ]

<transform element> ::=
    <to sql>
  | <from sql>

<to sql> ::=
  TO SQL WITH <to sql function>

<from sql> ::=
  FROM SQL WITH <from sql function>

<to sql function> ::=
  <specific routine designator>

<from sql function> ::=
  <specific routine designator>
```

Syntax Rules

1) Let *TD* be the <transform definition>. Let *DTN* be the <schema-resolved user-defined type name> immediately contained in *TD*. Let *DT* be the data type identified by *DTN*. Let *SDT* be the schema that includes the descriptor of *DT*. Let *TRD* be the transform descriptor included in the data type descriptor of *DT*.

2) No two <transform group>s immediately contained in *TD* shall have the same <group name>.

3) The SQL-invoked function identified by <to sql function> is called the *to-sql function*. The SQL-invoked function identified by <from sql function> is called the *from-sql function*.

4) Let *n* be the number of <transform group>s immediately contained in *TD*. For *i* ranging from 1 to *n*:

   a) Let TG_i be the *i*-th <transform group> immediately contained in *TD*. Let GN_i be the <group name> contained in TG_i.

b) Each of <to sql> and <from sql> immediately contained in TG_i shall be contained at most once in a <transform element list>.

c) The SQL-invoked routines identified by <to sql function> and <from sql function> shall be SQL-invoked functions that are deterministic and do not possibly modify SQL-data.

d) *TRD* shall not include a transform group descriptor *GD* that includes a group name that is equivalent to GN_i.

e) Let *SDTT* be the set that includes every data type DTT_j that is either a proper supertype or a proper subtype of *DT* such that the transform descriptor included in the data type descriptor of DTT_j includes a group descriptor $GDT_{j,k}$ that includes a group name that is equivalent to GN_i. *SDTT* shall be empty.

f) If <to sql> is specified, then let TSF_i be the SQL-invoked function identified by <to sql function>.

   i) Case:

      1) If TSF_i is an SQL-invoked method, then TSF_i shall have exactly two SQL parameters such that the declared type of the first SQL parameter is *DT* and the declared type of the second SQL parameter is a predefined data type. The result data type of TSF_i shall be *DT*.

      2) Otherwise, TSF_i shall have exactly one SQL parameter whose declared type is a predefined data type. The result data type of TSF_i shall be *DT*.

   ii) If *DT* is a structured type and TSF_i is an SQL-invoked method, then TSF_i shall be a type-preserving function.

g) If <from sql> is specified, then let FSF_i be the SQL-invoked function identified by <from sql function>. FSF_i shall have exactly one SQL parameter whose declared type is *DT*. The result data type of FSF_i shall be a predefined data type.

h) If <to sql> and <from sql> are both specified, then

   Case:

   i) If TSF_i is an SQL-invoked method, then the result data type of FSF_i and the data type of the second SQL parameter of TSF_i shall be compatible.

   ii) Otherwise, the result data type of FSF_i and the data type of the first SQL parameter of TSF_i shall be compatible.

Access Rules

1) For *i* ranging from 1 to *n*, the enabled authorization identifiers shall include the <authorization identifier> that owns *SDT* and the schema that includes the routine descriptors of TSF_i, if any, and FSF_i, if any.

General Rules

1) A <group name> specifies the group name that identifies a transform group.

2) For every TG_i, 1 (one) $\leq i \leq n$:

a) A new group descriptor GD_i is created that includes the <group name> immediately contained in TG_i. GD_i is included in the list of transform group descriptors included in *TRD*.

b) If <to sql> is specified, then the specific name of the to-sql function in GD_i is set to TSF_i.

c) If <from sql> is specified, then the specific name of the from-sql function in GD_i is set to FSF_i.

Conformance Rules

1) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <transform definition>.

11.58 <alter transform statement>

Function

Change the definition of one or more transform groups.

Format

```
<alter transform statement> ::=
  ALTER { TRANSFORM | TRANSFORMS }
      FOR <schema-resolved user-defined type name> <alter group>...

<alter group> ::=
  <group name> <left paren> <alter transform action list> <right paren>

<alter transform action list> ::=
  <alter transform action> [ { <comma> <alter transform action> }... ]

<alter transform action> ::=
    <add transform element list>
  | <drop transform element list>
```

Syntax Rules

1) Let *DN* be the <schema-resolved user-defined type name> and let *D* be the data type identified by *DN*. The schema identified by the explicit or implicit schema name of *DN* shall include the data type descriptor of *D*. Let *S* be that schema. Let *TD* be the transform descriptor included in the data type descriptor of *D*.

2) The scope of *DN* is the entire <alter transform statement> *AT*.

3) Let *n* be the number of <group name>s contained in *AT*. For *i* ranging from 1 to *n*:

   a) Let GN_i be the *i*-th <group name> contained in *AT*.

   b) For each GN_i, there shall be a transform group descriptor included in *TD* whose group name is equivalent to GN_i. Let GD_i be this transform group descriptor.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns *S*.

General Rules

1) For *i* ranging from 1 to *n*, GD_i is modified as specified by <alter transform action list>.

Conformance Rules

1) Without Feature S242, “Alter transform statement”, conforming SQL language shall not contain an <alter transform statement>.

11.59 <add transform element list>

Function

Add a transform element (<to sql> and/or <from sql>) to an existing transform group.

Format

```
<add transform element list> ::=
  ADD <left paren> <transform element list> <right paren>
```

Syntax Rules

1) Let *AD* be the <add transform element list>.

2) Let *DN* be the <schema-resolved user-defined type name> immediately contained in the containing <alter transform statement>. Let *D* be the user-defined type identified by *DN*. Let *TD* be the transform descriptor included in the data type descriptor of *D*.

3) Let *GD* be the transform group descriptor included in *TD* whose group name is equivalent to <group name> immediately contained in the containing <alter group>.

4) Each of <to sql> and <from sql> (immediately contained in *AD*) shall be contained at most once in the <transform element list>.

5) If *GD* includes a specific name of the to-sql function, then *AD* shall not contain <to sql>.

6) If *GD* includes a specific name of the from-sql function, then *AD* shall not contain <from sql>.

7) The SQL-invoked routine identified by either <to sql function> or <from sql function> shall be an SQL-invoked function that is deterministic and does not possibly modify SQL-data.

8) If <to sql> is specified, then let *TSF* be the SQL-invoked function identified by <to sql function>.

   a) Case:

      i) If *TSF* is an SQL-invoked method, then *TSF* shall have exactly two SQL parameters such that the declared type of the first SQL parameter is *D* and the declared type of the second SQL parameter is a predefined data type. The result data type of *TSF* shall be *D*.

      ii) Otherwise, *TSF* shall have exactly one SQL parameter whose declared type is a predefined data type. The result data type of *TSF* shall be *D*.

   b) If *D* is a structured type, then *TSF* shall be a type-preserving function.

   c) If *GD* includes the specific name of a from-sql function, then let *FS* be the SQL-invoked function that is identified by this specific name.

      Case:

      i) If *TSF* is an SQL-invoked method, then the result data type of *FS* and the data type of the second SQL parameter of *TSF* shall be compatible.

ii) Otherwise, the result data type of *FS* and the data type of the first SQL parameter of *TSF* shall be compatible.

9) If <from sql> is specified, then let *FSF* be the SQL-invoked function identified by <from sql function>.

a) *FSF* shall have exactly one SQL parameter whose declared type is *D*. The result data type of *FSF* shall be a predefined data type.

b) If *GD* includes the specific name of a to-sql function, then let *TS* be the SQL-invoked routine that is identified by this specific name.

Case:

i) If *TS* is an SQL-invoked method, then the result data type of *FSF* and the data type of the second SQL parameter of *TS* shall be compatible.

ii) Otherwise, the result data type of *FSF* and the data type of the first SQL parameter of *TS* shall be compatible.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema that includes the routine descriptors of *TSF*, if any, and *FSF*, if any.

General Rules

1) If <to sql> is specified, then the specific name of the to-sql function in *GD* is set to *TSF*.

2) If <from sql> is specified, then the specific name of the from-sql function in *GD* is set to *FSF*.

Conformance Rules

None.

11.60 <drop transform element list>

Function

Remove a transform element (<to sql> and/or <from sql>) from a transform group.

Format

```
<drop transform element list> ::=
  DROP <left paren> <transform kind>
      [ <comma> <transform kind> ] <drop behavior> <right paren>

<transform kind> ::=
    TO SQL
  | FROM SQL
```

Syntax Rules

1) Let *DN* be the <schema-resolved user-defined type name> immediately contained in the containing <alter transform statement>. Let *D* be the user-defined type identified by *DN*. Let *TD* be the transform descriptor included in the data type descriptor of *D*.

2) Let *GD* be the transform group descriptor included in *TD* whose group name is equivalent to <group name> immediately contained in the containing <alter group>.

3) Each of TO SQL and FROM SQL shall only be specified at most once in the <drop transform element list>.

4) If TO SQL is specified then *GD* shall include the specific name of a to-sql function. Let this function be *TSF*.

5) If FROM SQL is specified then *GD* shall include the specific name of a from-sql function. Let this function be *FSF*.

6) If RESTRICT is specified, then:

   a) If TO SQL is specified, then there shall be no external routine that has an SQL parameter whose associated to-sql function is *TSF* nor shall there be an external function that has *TSF* as the to-sql function associated with the result.

   b) If FROM SQL is specified, then there shall be no external routine that has an SQL parameter whose associated from-sql function is *FSF*.

Access Rules

None.

General Rules

1) If FROM SQL is specified, then:

a) Let *FSN* be the <specific name> of any external routine that has an SQL parameter whose associated from-sql function is *FSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE FSN CASCADE
```

b) The specific name of the from-sql function is removed from *GD*.

2) If TO SQL is specified, then:

a) Let *TSN* be the <specific name> of any external routine that has an SQL parameter whose associated to-sql function is *TSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE TSN CASCADE
```

b) Let *RSN* be the <specific name> of any external function that has *TSF* as the to-sql function associated with the result. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE RSN CASCADE
```

c) The specific name of the to-sql function is removed from *GD*.

Conformance Rules

None.

11.61 <drop transform statement>

Function

Remove one or more transform functions associated with a transform.

Format

```
<drop transform statement> ::=
  DROP { TRANSFORM | TRANSFORMS } <transforms to be dropped>
      FOR <schema-resolved user-defined type name> <drop behavior>

<transforms to be dropped> ::=
    ALL
  | <transform group element>

<transform group element> ::=
  <group name>
```

Syntax Rules

1) Let *DT* be the data type identified by <schema-resolved user-defined type name>. Let *SDT* be the schema that includes the descriptor of *DT*. Let *TRD* be the transform descriptor included in the data type descriptor of *DT*. Let *n* be the number of transform group descriptors in *TRD*.

2) If <transform group element> is specified, then *TRD* shall include a transform group descriptor *GD* that includes a group name that is equivalent to the <group name> immediately contained in <transform group element>.

3) If RESTRICT is specified, then

   Case:

   a) If ALL is specified, then for *i* ranging from 1 (one) to *n*:

      i) Let GD_i be the *i*-th transform group descriptor included in *TRD*.

      ii) If GD_i includes the specific name of a from-sql function FSF_i then there shall be no external routine that has an SQL parameter whose associated from-sql function is FSF_i.

      iii) If GD_i includes the specific name of a to-sql function TSF_i then there shall be no external routine that has an SQL parameter whose associated to-sql function is TSF_i nor shall there be an external function that has TSF_i as the to-sql function associated with the result.

   b) Otherwise:

      i) If *GD* includes the specific name of a from-sql function *FSF* then there shall be no external routine that has an SQL parameter whose associated from-sql function is *FSF*.

      ii) If *GD* includes the specific name of a to-sql function *TSF* then there shall be no external routine that has an SQL parameter whose associated to-sql function is *TSF* nor shall there be an external function that has *TSF* as the to-sql function associated with the result.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns *SDT*.

General Rules

1) Case:

   a) If ALL is specified, then, for *i* ranging from 1 (one) to *n*:

      i) Let GD_i be the *i*-th transform group descriptor included in *TRD*.

      ii) If GD_i includes the specific name of a from-sql function FSF_i, then let *FSN* be the <specific name> of any external routine that has an SQL parameter whose associated from-sql function is FSF_i. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE FSN CASCADE
```

      iii) If GD_i includes the specific name of a to-sql function TSF_i, then:

         1) Let *TSN* be the <specific name> of any external routine that has an SQL parameter whose associated to-sql function is TSF_i. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE TSN CASCADE
```

         2) Let *RSN* be the <specific name> of any external function that has TSF_i as the to-sql function associated with the result. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE RSN CASCADE
```

      iv) GD_i is removed from *TRD*.

   b) Otherwise:

      i) If *GD* includes the specific name of a from-sql function *FSF*, then let *FSN* be the <specific name> of any external routine that has an SQL parameter whose associated from-sql function is *FSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE FSN CASCADE
```

      ii) If *GD* includes the specific name of a to-sql function *TSF*, then:

         1) Let *TSN* be the <specific name> of any external routine that has an SQL parameter whose associated to-sql function is *TSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE TSN CASCADE
```

2) Let *RSN* be the <specific name> of any external function that has *TSF* as the to-sql function associated with the result. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE RSN CASCADE
```

iii) *GD* is removed from *TRD*.

Conformance Rules

1) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <drop transform statement>.

11.62 <sequence generator definition>

Function

Define an external sequence generator.

Format

```
<sequence generator definition> ::=
  CREATE SEQUENCE <sequence generator name> [ <sequence generator options> ]

<sequence generator options> ::=
  <sequence generator option> ...

<sequence generator option> ::=
    <sequence generator data type option>
  | <common sequence generator options>

<common sequence generator options> ::=
  <common sequence generator option> ...

<common sequence generator option> ::=
    <sequence generator start with option>
  | <basic sequence generator option>

<basic sequence generator option> ::=
    <sequence generator increment by option>
  | <sequence generator maxvalue option>
  | <sequence generator minvalue option>
  | <sequence generator cycle option>

<sequence generator data type option> ::=
  AS <data type>

<sequence generator start with option> ::=
  START WITH <sequence generator start value>

<sequence generator start value> ::=
  <signed numeric literal>

<sequence generator increment by option> ::=
  INCREMENT BY <sequence generator increment>

<sequence generator increment> ::=
  <signed numeric literal>

<sequence generator maxvalue option> ::=
    MAXVALUE <sequence generator max value>
  | NO MAXVALUE

<sequence generator max value> ::=
  <signed numeric literal>

<sequence generator minvalue option> ::=
    MINVALUE <sequence generator min value>
  | NO MINVALUE
```

```
<sequence generator min value> ::=
  <signed numeric literal>

<sequence generator cycle option> ::=
    CYCLE
  | NO CYCLE
```

Syntax Rules

1) Let *SEQ* be the sequence generator defined by the <sequence generator definition> *SEQD*.

2) If *SEQD* is contained in a <schema definition> *SD* and the <sequence generator name> *SQN* contains a <schema name>, then that <schema name> shall be equivalent to the implicit or explicit <schema name> of *SD*.

3) The schema identified by the explicit or implicit schema name of *SQN* shall not include a sequence generator descriptor whose sequence generator name is equivalent to *SQN*.

4) If *SEQD* is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>. Otherwise, let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *SQN*.

5) Each of <sequence generator data type option>, <sequence generator start with option>, <sequence generator increment by option>, <sequence generator maxvalue option>, <sequence generator minvalue option>, and <sequence generator cycle option> shall be specified at most once.

6) If <sequence generator data type option> is specified, then <data type> shall be an exact numeric type *DT* with scale 0 (zero); otherwise, let *DT* be an implementation-defined exact numeric type with scale 0 (zero).

7) The Syntax Rules of Subclause 9.24, "Creation of a sequence generator", are applied with <common sequence generator options> as *OPTIONS* and *DT* as *DATA TYPE*.

Access Rules

1) If a <sequence generator definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

General Rules

1) The General Rules of Subclause 9.24, "Creation of a sequence generator", are applied with <common sequence generator options> as *OPTIONS* and *DT* as *DATA TYPE*, yielding a sequence generator descriptor *SEQDS*. The sequence generator name included in *SEQDS* is set to *SQN*.

2) A privilege descriptor is created that defines the USAGE privilege on *SEQ* to *A*. This privilege is grantable. The grantor for this privilege descriptor is set to the special grantor value "_SYSTEM".

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <sequence generator definition>.

11.63 <alter sequence generator statement>

Function

Change the definition of an external sequence generator.

Format

```
<alter sequence generator statement> ::=
  ALTER SEQUENCE <sequence generator name> <alter sequence generator options>

<alter sequence generator options> ::=
  <alter sequence generator option>...

<alter sequence generator option> ::=
    <alter sequence generator restart option>
  | <basic sequence generator option>

<alter sequence generator restart option> ::=
  RESTART WITH <sequence generator restart value>

<sequence generator restart value> ::=
  <signed numeric literal>
```

Syntax Rules

1) Let *SEQ* be the sequence generator descriptor identified by the <sequence generator name> *SQN*. Let *DT* be the data type of *SEQ*.

2) The schema identified by the explicit or implicit schema name of *SQN* shall include *SEQ*.

3) The scope of *SQN* is the <alter sequence generator statement>.

4) The Syntax Rules of Subclause 9.25, "Altering a sequence generator", are applied with <alter sequence generator options> as *OPTIONS* and *SEQ* as *SEQUENCE*.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the explicit or implicit schema name of *SQN*.

General Rules

1) The General Rules of Subclause 9.25, "Altering a sequence generator", are applied with <alter sequence generator options> as *OPTIONS* and *SEQ* as *SEQUENCE*.

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain an <alter sequence generator statement>.

11.64 <drop sequence generator statement>

Function

Destroy an external sequence generator.

Format

```
<drop sequence generator statement> ::=
  DROP SEQUENCE <sequence generator name> <drop behavior>
```

Syntax Rules

1) Let *SEQ* be the sequence generator identified by the <sequence generator name> *SQN*.

2) The schema identified by the explicit or implicit schema name of *SQN* shall include the descriptor of *SEQ*.

3) If RESTRICT is specified, then *SEQ* shall not be referenced in any of the following:

   a) The SQL routine body of any routine descriptor.

   b) The trigger action of any trigger descriptor.

   NOTE 354 — If CASCADE is specified, then such referenced objects will be dropped by the execution of the <revoke statement> specified in the General Rules of this Subclause.

4) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the sequence generator identified by *SQN*.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

   REVOKE USAGE ON SEQUENCE *SQN* FROM *A* CASCADE

2) The descriptor of *SEQ* is destroyed.

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <drop sequence generator statement>.

12 Access control

This Clause is modified by Clause 10, "Access control", in ISO/IEC 9075-4.
This Clause is modified by Clause 13, "Access control", in ISO/IEC 9075-9.
This Clause is modified by Clause 10, "Access control", in ISO/IEC 9075-13.

12.1 <grant statement>

This Subclause is modified by Subclause 10.1, "<grant statement>", in ISO/IEC 9075-4.

Function

Define privileges and role authorizations.

Format

```
<grant statement> ::=
    <grant privilege statement>
  | <grant role statement>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) For every involved grantee *G* and for every domain *D1* owned by *G*, if all of the following are true:

   a) The applicable privileges for *G* include the grantable REFERENCES privilege on every column referenced in the <search condition> *SC* included in a domain constraint descriptor included in the domain descriptor of *D1*.

   b) The applicable privileges for *G* include the grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *SC*.

   c) The applicable privileges for *G* include the grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *SC* such that *T1* is in the scope of

the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR* included in a domain constraint descriptor included in the domain descriptor of *D1*.

d) The applicable privileges for *G* include the grantable SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of every <reference resolution> contained in *SC*.

e) The applicable privileges for *G* include the grantable USAGE privilege on all domains, character sets, collations, and transliterations whose <domain name>s, <character set name>s, <collation name>s, and <transliteration name>s, respectively, are included in the domain descriptor of *D1*.

then for every privilege descriptor with <action> USAGE, a grantor of "_SYSTEM", object *D1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed with a current user identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT USAGE ON DOMAIN D1
TO G WITH
GRANT OPTION
```

2) For every involved grantee *G* and for every collation *C1* owned by *G*, if the applicable privileges for *G* include a grantable USAGE privilege for the character set name included in the collation descriptor of *C1* and a grantable USAGE privilege for the transliteration name, if any, included in the collation descriptor of *C1*, then for every privilege descriptor with <action> USAGE, a grantor of "_SYSTEM", object of *C1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed with a current user identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT USAGE ON
  COLLATION C1
  TO G
  WITH GRANT OPTION
```

3) For every involved grantee *G* and for every transliteration *T1* owned by *G*, if the applicable privileges for *G* contain a grantable USAGE privilege for every character set identified by a <character set specification> contained in the <transliteration definition> of *T1*, then for every privilege descriptor with <action> *P*, a grantor of "_SYSTEM", object of *T1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT P
    ON TRANSLATION T1
    TO G
    WITH GRANT OPTION
```

4) For every table *T* specified by some involved privilege descriptor and for each view *V* owned by some involved grantee *G* such that *T* or some column *CT* of *T* is referenced in the <query expression> *QE* of *V*, or *T* is a supertable of the scoped table of a <reference resolution> contained in *QE*, let RT_i, for *i* ranging from 1 (one) to the number of tables identified by the <table reference>s contained in *QE*, be the <table name>s of those tables. For every column *CV* of *V*:

a) Let $CRT_{i,j}$, for *j* ranging from 1 (one) to the number of columns of RT_i that are underlying columns of *CV*, be the <column name>s of those columns.

b) If, following successful execution of the <grant statement>, all of the following are true:

i) The applicable privileges for *G* include grantable SELECT privileges on all of the columns $CRT_{i,j}$.

ii) The applicable privileges for *G* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

iii) The applicable privileges for *G* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference>. *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

iv) The applicable privileges for *G* include grantable SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

then the following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT SELECT (CV)
    ON V
    TO G
    WITH GRANT OPTION
```

c) If, following successful execution of the <grant statement>, the applicable privileges for *G* will include REFERENCES($CRT_{i,j}$) for all *i* and for all *j*, and will include a REFERENCES privilege on some column of RT_i for all *i*, then:

i) Case:

1) If all of the following are true, then let *WGO* be "WITH GRANT OPTION".

A) The applicable privileges for *G* will include grantable REFERENCES($CRT_{i,j}$) for all *i* and for all *j*, and will include a grantable REFERENCES privilege on some column of RT_i for all *i*.

B) The applicable privileges for *G* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

C) The applicable privileges for *G* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference>. *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

D) The applicable privileges for *G* include grantable SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

2) Otherwise, let *WGO* be a zero-length string.

ii) The following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT REFERENCES (CV)
    ON V
```

```
        TO G
        WGO
```

d) If, following successful execution of the <grant statement>, the applicable privileges for *G* include grantable SELECT privilege on every column of *V*, then the following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT SELECT
    ON V
    TO G
    WITH GRANT OPTION
```

e) Following successful execution of the <grant statement>,

Case:

i) If the applicable privileges for *G* include grantable REFERENCES privilege on every column of *V*, then let *WGO* be "WITH GRANT OPTION".

ii) If the applicable privileges for *G* include REFERENCES privilege on every column of *V*, then let *WGO* be a zero-length string.

The following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT REFERENCES
    ON V
    TO G
    WGO
```

5) Following the successful execution of the <grant statement>, for every table *T* specified by some involved privilege descriptor and for every effectively updatable view *V* owned by some grantee *G* such that *T* is some leaf underlying table of the <query expression> of *V*, the General Rules of Subclause 10.11, "Determination of view and view component privileges", are applied with *V* as the *VIEW*.

6) For every involved grantee *G* and for every referenceable view *V*, named *VN*, owned by *G*, if following the successful execution of the <grant statement>, the applicable privileges for *G* include grantable UNDER privilege on the direct supertable of *V*, then the following <grant statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT UNDER
    ON VN
    TO G
    WITH GRANT OPTION
```

7) For every involved grantee *G* and for every schema-level SQL-invoked routine *R1* owned by *G*, if the applicable privileges for *G* contain all of the privileges necessary to successfully execute every <SQL procedure statement> contained in the <routine body> of *R1* are grantable, then for every privilege descriptor with <action> EXECUTE, a grantor of "_SYSTEM", object of *R1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT EXECUTE
    ON R1
```

```
TO G
WITH GRANT OPTION
```

NOTE 355 — The privileges necessary include the EXECUTE privilege on every subject routine of every <routine invocation> contained in the <SQL procedure statement>.

8) If two privilege descriptors are identical except that one indicates that the privilege is grantable and the other indicates that the privilege is not grantable, then both privilege descriptors are set to indicate that the privilege is grantable.

9) If two privilege descriptors are identical except that one indicates WITH HIERARCHY OPTION and the other does not, then both privilege descriptors are set to indicate that the privilege has the WITH HIERARCHY OPTION.

10) Redundant duplicate privilege descriptors are removed from the collection of all privilege descriptors.

11) Redundant duplicate view privilege dependency descriptors are removed from the collection of all view privilege dependency descriptors.

Conformance Rules

None.

12.2 <grant privilege statement>

This Subclause is modified by Subclause 10.1, "<grant privilege statement>", in ISO/IEC 9075-13.

Function

Define privileges.

Format

```
<grant privilege statement> ::=
  GRANT <privileges> TO <grantee> [ { <comma> <grantee> }... ]
      [ WITH HIERARCHY OPTION ]
      [ WITH GRANT OPTION ]
      [ GRANTED BY <grantor> ]
```

Syntax Rules

1) Let *O* be the object identified by the <object name> contained in <privileges>.

2) The schema identified by the explicit or implicit qualifier of the <object name> shall include the descriptor of *O*.

3) If WITH HIERARCHY OPTION is specified, then:

   a) <privileges> shall specify an <action> of SELECT without a <privilege column list> and without a <privilege method list>.

   b) *O* shall be a table of a structured type.

Access Rules

None.

General Rules

1) Case:

   a) If GRANTED BY is omitted, then let *G* be OMITTED.

   b) Otherwise, let *G* be <grantor>.

2) Let *A* be the result of applying the General Rules of Subclause 12.8, "Grantor determination", with *G* as *GRANTOR*.

3) If the applicable privileges for *A* do not include a privilege identifying *O*, then an exception condition is raised: *privilege not granted*.

4) A set of privilege descriptors is identified. The privilege descriptors identified are those defining, for each <action> explicitly or implicitly in <privileges>, that <action> on *O* held by *A* with grant option.

5) The <object privileges> specify one or more privileges on the object identified by the <object name>.

6) For every identified privilege descriptor *IPD*, a privilege descriptor is created for each <grantee>, that specifies grantee <grantee>, action <action>, object *O*, and grantor *A*. Let *CPD* be the set of privilege descriptors created.

7) For every privilege descriptor in *CPD* whose action is INSERT, UPDATE, or REFERENCES without a column name, privilege descriptors are also created and added to *CPD* for each column *C* in *O* for which *A* holds the corresponding privilege with grant option. For each such column, a privilege descriptor is created that specifies the identical <grantee>, the identical <action>, object *C*, and grantor *A*.

8) For every privilege descriptor in *CPD* whose action is SELECT without a column name or method name, privilege descriptors are also created and added to *CPD* for each column *C* in *O* for which *A* holds the corresponding privilege with grant option. For each such column, a privilege descriptor is created that specifies the identical <grantee>, the identical <action>, object *C*, and grantor *A*.

9) For every privilege descriptor in *CPD* whose action is SELECT without a column name or method name, if the table *T* identified by the object of the privilege descriptor is a table of a structured type *TY*, then table/method privilege descriptors are also created and added to *CPD* for each method *M* of *TY* for which *A* holds the corresponding privilege with grant option. For each such method, a table/method privilege descriptor is created that specifies the identical <grantee>, the identical <action>, object consisting of the pair of table *T* and method *M*, and grantor *A*.

10) If WITH GRANT OPTION was specified, then each privilege descriptor also indicates that the privilege is grantable.

11) Let *SWH* be the set of privilege descriptors in *CPD* whose action is SELECT WITH HIERARCHY OPTION. Let *ST* be the set of subtables of *O*. For every table *T* in *ST* and for every privilege descriptor in *SWH* grantee *G*, and grantor *A*,

    Case:

    a) If the privilege is grantable, then let *WGO* be “WITH GRANT OPTION”.

    b) Otherwise, let *WGO* be the zero-length string.

    The following <grant statement> is effectively executed without further Access Rule checking:

```
GRANT SELECT
    ON T
    TO G
    WGO
    GRANTED BY A
```

12) For every combination of <grantee> and <action> on *O* specified in <privileges>, if there is no corresponding privilege descriptor in *CPD*, then a completion condition is raised: *warning — privilege not granted*.

13) If ALL PRIVILEGES was specified, then for each grantee *G*, if there is no privilege descriptor in *CPD* specifying grantee *G*, then a completion condition is raised: *warning — privilege not granted*.

14) The *set of involved privilege descriptors* is defined to be *CPD*.

15) The *set of involved grantees* is defined as the set of specified <grantee>s.

Conformance Rules

1) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <specific routine designator> contained in a <grant privilege statement> that identifies a method.

2) Without Feature S081, “Subtables”, conforming SQL language shall not contain a <grant privilege statement> that contains WITH HIERARCHY OPTION.

12.3 <privileges>

This Subclause is modified by Subclause 10.2, "<privileges>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 13.1, "<privileges>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.2, "<privileges>", in ISO/IEC 9075-13.

Function

Specify privileges.

Format

```
<privileges> ::=
  <object privileges> ON <object name>

<object name> ::=
    [ TABLE ] <table name>
  |DOMAIN <domain name>
  | COLLATION <collation name>
  | CHARACTER SET <character set name>
  | TRANSLATION <transliteration name>
  | TYPE <schema-resolved user-defined type name>
  | SEQUENCE <sequence generator name>
  | <specific routine designator>

<object privileges> ::=
    ALL PRIVILEGES
  | <action> [ { <comma> <action> }... ]

<action> ::=
    SELECT
  | SELECT <left paren> <privilege column list> <right paren>
  | SELECT <left paren> <privilege method list> <right paren>
  | DELETE
  | INSERT [ <left paren> <privilege column list> <right paren> ]
  | UPDATE [ <left paren> <privilege column list> <right paren> ]
  | REFERENCES [ <left paren> <privilege column list> <right paren> ]
  | USAGE
  | TRIGGER
  | UNDER
  | EXECUTE

<privilege method list> ::=
  <specific routine designator> [ { <comma> <specific routine designator> }... ]

<privilege column list> ::=
  <column name list>

<grantee> ::=
    PUBLIC
  | <authorization identifier>

<grantor> ::=
    CURRENT_USER
```

```
| CURRENT_ROLE
```

Syntax Rules

1) ALL PRIVILEGES is equivalent to the specification of all of the privileges on <object name> for which the <grantor> has grantable privilege descriptors.

2) If the <object name> of the <grant statement> or <revoke statement> specifying <privileges> specifies <table name>, then let *T* be the table identified by that <table name>. *T* shall not be a declared local temporary table.

3) If <object name> specifies a <domain name>, <collation name>, <character set name>, <transliteration name>, <schema-resolved user-defined type name>, or <sequence generator name>, then <privileges> may specify USAGE. Otherwise, USAGE shall not be specified.

4) If <object name> specifies a <table name> that identifies a base table, then <privileges> may specify TRIGGER; otherwise, TRIGGER shall not be specified.

5) If <object name> specifies a <schema-resolved user-defined type name> that identifies a structured type or specifies a <table name>, then <privileges> may specify UNDER; otherwise, UNDER shall not be specified.

6) If *T* is a temporary table, then <privileges> shall specify ALL PRIVILEGES.

7) If the object identified by <object name> of the <grant statement> or <revoke statement> is an SQL-invoked routine, then <privileges> may specify EXECUTE; otherwise, EXECUTE shall not be specified.

8) The <object privileges> specify one or more privileges on the object identified by <object name>.

9) Each <column name> in a <privilege column list> shall identify a column of *T*.

10) If <privilege method list> is specified, then <object name> shall specify a <table name> that identifies a table of a structured type *TY* and each <specific routine designator> in the <privilege method list> shall identify a method of *TY*.

11) UPDATE (<privilege column list>) is equivalent to the specification of UPDATE (<column name>) for each <column name> in <privilege column list>. INSERT (<privilege column list>) is equivalent to the specification of INSERT (<column name>) for each <column name> in <privilege column list>. REFERENCES (<privilege column list>) is equivalent to the specification of REFERENCES (<column name>) for each <column name> in <privilege column list>. SELECT (<privilege column list>) is equivalent to the specification of SELECT (<column name>) for each <column name> in <privilege column list>. SELECT (<privilege method list>) is equivalent to the specification of SELECT (<specific routine designator>) for each <specific routine designator> in <privilege method list>.

Access Rules

None.

General Rules

1) A <grantee> of PUBLIC denotes at all times a list of <grantee>s containing all of the <authorization identifier>s in the SQL-environment.

2) SELECT (<column name>) specifies the SELECT privilege on the indicated column and implies one or more column privilege descriptors.

3) SELECT (<specific routine designator>) specifies the SELECT privilege on the indicated method for the table identified by <object name> and implies one or more table/method privilege descriptors.

4) SELECT with neither <privilege column list> nor <privilege method list> specifies the SELECT privilege on all columns of *T* including any columns subsequently added to *T* and implies a table privilege descriptor and one or more column privilege descriptors. If *T* is a table of a structured type *TY*, then SELECT also specifies the SELECT privilege on all methods of the type *TY*, including any methods subsequently added to the type *TY*, and implies one or more table/method privilege descriptors.

5) UPDATE (<column name>) specifies the UPDATE privilege on the indicated column and implies one or more column privilege descriptors. If the <privilege column list> is omitted, then UPDATE specifies the UPDATE privilege on all columns of *T*, including any column subsequently added to *T* and implies a table privilege descriptor and one or more column privilege descriptors.

6) INSERT (<column name>) specifies the INSERT privilege on the indicated column and implies one or more column privilege descriptors. If the <privilege column list> is omitted, then INSERT specifies the INSERT privilege on all columns of *T*, including any column subsequently added to *T* and implies a table privilege descriptor and one or more column privilege descriptors.

7) REFERENCES (<column name>) specifies the REFERENCES privilege on the indicated column and implies one or more column privilege descriptors. If the <privilege column list> is omitted, then REFERENCES specifies the REFERENCES privilege on all columns of *T*, including any column subsequently added to *T* and implies a table privilege descriptor and one or more column privilege descriptors.

8) *B* has the WITH ADMIN OPTION on a role if a role authorization descriptor identifies the role as granted to *B* WITH ADMIN OPTION or a role authorization descriptor identifies it as granted WITH ADMIN OPTION to another applicable role for *B*.

Conformance Rules

1) Without Feature T332, “Extended roles”, conforming SQL language shall not contain a <grantor>.

2) Without Feature T211, “Basic trigger capability”, conforming SQL language shall not contain an <action> that contains TRIGGER.

3) Without Feature S081, “Subtables”, conforming SQL language shall not contain a <privileges> that contains an <action> that contains UNDER and that contains an <object name> that contains a <table name>.

4) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <privileges> that contains an <action> that contains UNDER and that contains an <object name> that contains a <schema-resolved user-defined type name> that identifies a structured type.

5) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <privileges> that contains an <action> that contains USAGE and that contains an <object name> that contains a <schema-resolved user-defined type name> that identifies a structured type.

6) Without Feature T281, “SELECT privilege with column granularity”, in conforming SQL language, an <action> that contains SELECT shall not contain a <privilege column list>.

7) Without Feature F731, “INSERT column privileges”, in conforming SQL language, an <action> that contains INSERT shall not contain a <privilege column list>.

8) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <privilege method list>.

12.4 <role definition>

Function

Define a role.

Format

```
<role definition> ::=
  CREATE ROLE <role name> [ WITH ADMIN <grantor> ]
```

Syntax Rules

1) The specified <role name> shall not be equivalent to any other <authorization identifier> in the SQL-environment.

Access Rules

1) The privileges necessary to execute the <role definition> are implementation-defined.

General Rules

1) A role descriptor whose role name is <role name> is created in the SQL-environment.

2) Case:

   a) If WITH ADMIN is omitted, then let *G* be OMITTED.

   b) Otherwise, let *G* be <grantor>.

3) Let *A* be the result of applying the General Rules of Subclause 12.8, "Grantor determination", with *G* as *GRANTOR*.

4) A grantable role authorization descriptor is created whose role name is <role name>, whose grantor of "_SYSTEM", and whose grantee is *A*.

Conformance Rules

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

2) Without Feature T332, "Extended roles", conforming SQL language shall not contain a <role definition> that immediately contains WITH ADMIN.

12.5 <grant role statement>

Function

Define role authorizations.

Format

```
<grant role statement> ::=
  GRANT <role granted> [ { <comma> <role granted> }... ]
      TO <grantee> [ { <comma> <grantee> }... ]
      [ WITH ADMIN OPTION ]
      [ GRANTED BY <grantor> ]

<role granted> ::=
  <role name>
```

Syntax Rules

1) No role identified by a specified <grantee> shall be applicable for any role identified by a specified <role granted>.

   NOTE 356 — That is, no cycles of role authorizations are allowed.

Access Rules

None.

General Rules

1) Case:

   a) If WITH ADMIN is omitted, then let *G* be OMITTED.

   b) Otherwise, let *G* be <grantor>.

2) Let *A* be the result of applying the General Rules of Subclause 12.8, "Grantor determination", with *G* as *GRANTOR*.

3) For each <role granted> *R*, if no grantable role authorization descriptor exists whose role name is *R* and whose grantee is *A* or an applicable role for *A*, then an exception condition is raised: *invalid role specification*.

4) For each <grantee> *GEE*, for each <role granted> *R*, a role authorization descriptor is created with role name *R*, grantee *GEE*, and grantor *A*.

5) If WITH ADMIN OPTION is specified, then each role authorization descriptor is grantable.

6) If two role authorization descriptors are identical except that one is grantable and the other is not, then both role authorization descriptors are set to indicate that the role authorization is grantable.

7) Redundant duplicate role authorization descriptors are destroyed.

8) The *set of involved privilege descriptors* is the union of the sets of privilege descriptors corresponding to the applicable privileges for every <role granted> specified.

9) The *set of involved grantees* is the union of the set of <grantee>s and the set of <role name>s for which at least one of the <role name>s that is possibly specified as a <grantee> is applicable.

Conformance Rules

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

12.6 <drop role statement>

Function

Destroy a role.

Format

```
<drop role statement> ::=
  DROP ROLE <role name>
```

Syntax Rules

1) Let *R* be the role identified by the specified <role name>.

Access Rules

1) There shall exist at least one grantable role authorization descriptor whose role name is *R* and whose grantee is an enabled authorization identifier.

General Rules

1) Let *A* be any <authorization identifier> identified by a role authorization descriptor as having been granted to *R*.

2) The following <revoke role statement> is effectively executed without further Access Rule checking:

   ```
   REVOKE R FROM A
   ```

3) The descriptor of *R* is destroyed.

Conformance Rules

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

12.7 <revoke statement>

This Subclause is modified by Subclause 10.3, "<revoke statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 13.2, "<revoke statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.3, "<revoke statement>", in ISO/IEC 9075-13.

Function

Destroy privileges and role authorizations.

Format

```
<revoke statement> ::=
    <revoke privilege statement>
  | <revoke role statement>

<revoke privilege statement> ::=
  REVOKE [ <revoke option extension> ] <privileges>
      FROM <grantee> [ { <comma> <grantee> }... ]
      [ GRANTED BY <grantor> ]
      <drop behavior>

<revoke option extension> ::=
    GRANT OPTION FOR
  | HIERARCHY OPTION FOR

<revoke role statement> ::=
  REVOKE [ ADMIN OPTION FOR ] <role revoked> [ { <comma> <role revoked> }... ]
      FROM <grantee> [ { <comma> <grantee> }... ]
      [ GRANTED BY <grantor> ]
      <drop behavior>

<role revoked> ::=
  <role name>
```

Syntax Rules

1) Let *O* be the object identified by the <object name> contained in <privileges>. If *O* is a table *T*, then let *S* be the set of subtables of *O*. If *T* is a table of a structured type, then let *TY* be that type.

2) If WITH HIERARCHY OPTION is specified, the <privileges> shall specify an <action> of SELECT without a <privilege column list> and without a <privilege method list> and *O* shall be a table of a structured type.

3) SELECT is equivalent to specifying both the SELECT table privilege and SELECT (<privilege column list>) for all columns of <table name>. If *T* is a table of a structured type *TY*, then SELECT also specifies SELECT (<privilege column list>) for all columns inherited from *T* in each of the subtables of *T*, and SELECT (<privilege method list>) for all methods of *TY* in each of the subtables of *T*.

4) INSERT is equivalent to specifying both the INSERT table privilege and INSERT (<privilege column list>) for all columns of <table name>.

5) UPDATE is equivalent to specifying both the UPDATE table privilege and UPDATE (<privilege column list>) for all columns of <table name>, as well as UPDATE (<privilege column list>) for all columns inherited from *T* in each of the subtables of *T*.

6) REFERENCES is equivalent to specifying both the REFERENCES table privilege and REFERENCES (<privilege column list>) for all columns of <table name>, as well as REFERENCES (<privilege column list>) for all columns inherited from *T* in each of the subtables of *T*.

Access Rules

None.

General Rules

1) Case:

   a) If GRANTED BY is omitted, then let *G* be OMITTED.

   b) Otherwise, let *G* be <grantor>.

2) Let *A* be the result of applying the General Rules of Subclause 12.8, "Grantor determination", with *G* as *GRANTOR*.

3) Case:

   a) If the <revoke statement> is a <revoke privilege statement>, and the applicable privileges for *A* do not include a privilege identifying *O*, then an exception condition is raised: *invalid grantor*.

   b) If the <revoke statement> is a <revoke role statement>, then, for every role *R* identified by a <role revoked>, if the applicable roles of *A* do not include a role *AR* such that there exists a grantable role authorization descriptor with role *R* and grantee *A*, then an exception condition is raised: *invalid grantor*.

4) Case:

   a) If the <revoke statement> is a <revoke privilege statement>, then, for every <grantee> specified, a set of privilege descriptors is identified. A privilege descriptor *P* is said to be *identified* if it belongs to the set of privilege descriptors that defined, for any <action> explicitly or implicitly in <privileges>, that <action> on *O*, or any of the objects in *S*, granted by *A* to <grantee>.

      NOTE 357 — Column privilege descriptors become identified when <action> explicitly or implicitly contains a <privilege column list>. Table/method descriptors become identified when <action> explicitly or implicitly contains a <privilege method list>.

   b) If the <revoke statement> is a <revoke role statement>, then, for every <grantee> specified, a set of role authorization descriptors is identified. A role authorization descriptor is said to be *identified* if it defines the grant of any of the specified <role revoked>s to <grantee> with grantor *A*.

5) A privilege descriptor *D* is said to be *directly dependent* on another privilege descriptor *P* if

   Case:

   a) If *D* identifies a view or a column of a view and the action of *D* is INSERT, UPDATE, or DELETE, then *D* is *directly dependent* on *P* if there exists a view privilege dependency descriptor whose supporting privilege descriptor is *P* and whose dependent privilege descriptor is *D*.

b) Otherwise, *D* is directly dependent on *P* if one of the following is true:

i) All of the following conditions hold:

1) *P* indicates that the privilege that it represents is grantable.

2) The grantee of *P* is the same as the grantor of *D*, or the grantee of *P* is PUBLIC, or, if the grantor of *D* is a <role name>, the grantee of *P* is an applicable role for the grantor of *D*.

3) Case:

A) *P* and *D* are both column privilege descriptors. The action and the identified column of *P* are the same as the action and identified column of *D*, respectively.

B) *P* and *D* are both table privilege descriptors. The action and the identified table of *P* are the same as the action and the identified table of *D*, respectively.

C) *P* and *D* are both execute privilege descriptors. The action and the identified SQL-invoked routine of *P* are the same as the action and the identified SQL-invoked routine of *D*, respectively.

D) *P* and *D* are both usage privilege descriptors. The action and the identified domain, character set, collation, transliteration, user-defined type, or sequence generator of *P* are the same as the action and the identified domain, character set, collation, transliteration, user-defined type, or sequence generator of *D*, respectively.

E) *P* and *D* are both under privilege descriptors. The action and the identified user-defined type or table of *P* are the same as the action and the identified user-defined type or table of *D*, respectively.

F) *P* and *D* are both table/method privilege descriptors. The action and the identified method and table of *P* are the same as the action and the identified method and table of *D*, respectively.

ii) All of the following conditions hold:

1) The privilege descriptor for *D* indicates that its grantor is the special grantor value "_SYSTEM".

2) The action of *P* is the same as the action of *D*.

3) The grantee of *P* is the owner of the table, collation, or transliteration identified by *D* or the grantee of *P* is PUBLIC.

4) One of the following conditions hold:

A) *P* and *D* are both column privilege descriptors, the privilege descriptor *D* identifies a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V*, and one of the following is true:

I) For every table *T* identified by a <table reference> contained in the <query expression> of *V* and for every column *CT* that is a column of *T* and an underlying column of *CV*, the action for *P* is REFERENCES and either the identified column of *P* is *CT* or the identified table of *P* is *T*.

II) For every table *T* identified by a <table reference> contained in the <query expression> of *V* and for every column *CT* that is a column of *T* and an underlying

column of *CV*, the action for *P* is SELECT and either the identified column of *P* is *CT* or the identified table of *P* is *T*.

B) The privilege descriptor *D* identifies the <collation name> of a <collation definition> *CO* and the identified character set name of *P* is included in the collation descriptor for *CO*, or the identified transliteration name of *P* is included in the collation descriptor for *CO*.

C) The privilege descriptor *D* identifies the <transliteration name> of a <transliteration definition> *TD* and the identified character set name of *P* is contained in the <source character set specification> or the <target character set specification> immediately contained in *TD*.

iii) All of the following conditions hold:

1) The privilege descriptor for *D* indicates that its grantor is the special grantor value "_SYSTEM".

2) The grantee of *P* is the owner of the domain identified by *D* or the grantee of *P* is PUBLIC.

3) The privilege descriptor *D* identifies the <domain name> of a <domain definition> *DO* and either the column privilege descriptor *P* has an action of REFERENCES and identifies a column referenced in the <search condition> included in the domain descriptor for *DO*, or the privilege descriptor *P* has an action of USAGE and identifies a domain, collation, character set, or transliteration whose <domain name>, <collation name>, <character set name>, or <transliteration name>, respectively, is contained in the <search condition> of the domain descriptor for *DO*.

6) The *privilege dependency graph* is a directed graph such that all of the following are true:

a) Each node represents a privilege descriptor.

b) Each arc from node *P1* to node *P2* represents the fact that *P2* directly depends on *P1*.

An *independent node* is a node that has no incoming arcs.

7) A privilege descriptor *P* is said to be *modified* if all of the following are true:

a) *P* indicates that the privilege that it represents is grantable.

b) *P* directly depends on an identified privilege descriptor or a modified privilege descriptor.

c) Case:

i) If *P* is neither a SELECT nor a REFERENCES column privilege descriptor that identifies a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V*, then let *XO* and *XA* respectively be the identifier of the object identified by a privilege descriptor *X* and the action of *X*. Within the set of privilege descriptors upon which *P* directly depends, there exist some *XO* and *XA* for which the set of identified privilege descriptors unioned with the set of modified privilege descriptors include all privilege descriptors specifying the grant of *XA* on *XO* WITH GRANT OPTION.

ii) If *P* is a column privilege descriptor that identifies a column *CV* identified by a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V* with an action *PA* of REFERENCES or SELECT, then let *SP* be the set of privileges upon which *P* directly depends. For every table *T* identified by a <table reference> contained in the <query

expression> of *V*, let *RT* be the <table name> of *T*. There exists a column *CT* whose <column name> is *CRT*, such that all of the following are true:

1) *CT* is a column of *T* and an underlying column of *CV*.

2) Every privilege descriptor *PD* that is the descriptor of some member of *SP* that specifies the action *PA* on *CRT* WITH GRANT OPTION is either an identified privilege descriptor for *CRT* or a modified privilege descriptor for *CRT*.

d) At least one of the following is true:

i) GRANT OPTION FOR is specified and the grantor of *P* is the special grantor value "_SYSTEM".

ii) There exists a path to *P* from an independent node that includes no identified or modified privilege descriptors. *P* is said to be a *marked modified privilege descriptor*.

iii) *P* directly depends on a marked modified privilege descriptor, and the grantor of *P* is the special grantor value "_SYSTEM". *P* is said to be a *marked modified privilege descriptor*.

8) A role authorization descriptor *D* is said to be *directly dependent* on another role authorization descriptor *RD* if all of the following conditions hold:

a) *RD* indicates that the role that it represents is grantable.

b) The role name of *D* is the same as the role name of *RD*.

c) The grantee of *RD* is the same as the grantor of *D*, or the grantee of *RD* is PUBLIC, or, if the grantor of *D* is a <role name>, the grantee of *RD* is an applicable role for the grantor of *D*.

9) The *role dependency graph* is a directed graph such that all of the following are true:

a) Each node represents a role authorization descriptor.

b) Each arc from node *R1* to node *R2* represents the fact that *R2* directly depends on *R1*.

An independent node is one that has no incoming arcs.

10) A role authorization descriptor *RD* is said to be *abandoned* if it is not an independent node, and it is not itself an identified role authorization descriptor, and there exists no path to *RD* from any independent node other than paths that include an identified role authorization descriptor.

11) An arc from a node *P* to a node *D* of the privilege dependency graph is said to be *unsupported* if all of the following are true:

a) The grantor of *D* and the grantee of *P* are both <role name>s.

b) The destruction of all abandoned role authorization descriptors and, if ADMIN OPTION FOR is not specified, all identified role authorization descriptors would result in the grantee of *P* no longer being an applicable role for the grantor of *D*.

12) A privilege descriptor *P* is *abandoned* if

Case:

a) It is not an independent node, and *P* is not itself an identified or a modified privilege descriptor, and there exists no path to *P* from any independent node other than paths that include an identified privilege descriptor or a modified privilege descriptor or an unsupported arc, and, if <revoke statement> specifies WITH HIERARCHY OPTION, then *P* has the WITH HIERARCHY OPTION.

b) All of the following conditions hold:

i) *P* is a column privilege descriptor that identifies a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V*, with an action *PA* of REFERENCES or SELECT.

ii) Letting *SP* be the set of privileges upon which *P* directly depends, at least one of the following is true:

1) There exists some table name *RT* such that all of the following are true:

A) *RT* is the name of the table identified by some <table reference> contained in the <query expression> of *V*.

B) For every column privilege descriptor *CPD* that is the descriptor of some member of *SP* that specifies the action *PA* on *RT*, *CPD* is either an identified privilege descriptor for *RT* or an abandoned privilege descriptor for *RT*.

2) There exists some column name *CRT* such that all of the following are true:

A) *CRT* is the name of some column of the table identified by some <table reference> contained in the <query expression> of *V*.

B) For every column privilege descriptor *CPD* that is the descriptor of some member of *SP* that specifies the action *PA* on *CRT*, *CPD* is either an identified privilege descriptor for *CRT* or an abandoned privilege descriptor for *CRT*.

13) The *revoke destruction action* is defined as

Case:

a) If the <revoke statement> is a <revoke privilege statement>, then

Case:

i) If the <revoke statement> specifies the WITH HIERARCHY OPTION, then the removal of the WITH HIERARCHY OPTION from all identified and abandoned privilege descriptors.

ii) Otherwise, the destruction of all abandoned privilege descriptors and, if GRANT OPTION FOR is not specified, all identified privilege descriptors.

b) If the <revoke statement> is a <revoke role statement>, then the destruction of all abandoned role authorization descriptors, all abandoned privilege descriptors and, if GRANT ADMIN FOR is not specified, all identified role authorization descriptors.

14) Let *S1* be the name of any schema and *A1* be the <authorization identifier> that owns the schema identified by *S1*.

15) Let *V* be any view descriptor included in *S1*. Let *QE* be the <query expression> of *V*. *V* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

a) SELECT privilege on at least one column of every table identified by a <table reference> is contained in *QE*.

b) SELECT privilege on every column identified by a <column reference> contained in *QE*.

c) USAGE privilege on every domain, every collation, every character set, and every transliteration whose names are contained in *QE*.

d) USAGE privilege on any user-defined type *UDT* such that some <data type> contained in *V* is usage-dependent on *UDT*.

e) EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in *QE*.

f) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is subject routine of *MR*.

g) SELECT privilege on any column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in *QE*.

h) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of any <reference resolution> that is contained in *QE*.

i) SELECT privilege on the scoped table of any <reference resolution> that is contained in *QE*.

j) If *V* is the descriptor of a referenceable table, then USAGE privilege on the structured type associated with the view described by *V*.

k) UNDER privilege on every direct supertable of the view described by *V*.

l) SELECT privilege WITH HIERARCHY OPTION privilege on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in *QE*.

16) Let *T* be any table descriptor included in *S1*. *T* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having any of the following:

a) If *T* is the descriptor of a referenceable table, then USAGE privilege on the structured type associated with the table described by *T*.

b) UNDER privilege on every direct supertable of the table described by *T*.

17) Let *TC* be any table constraint descriptor included in *S1*. *TC* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

a) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in the applicable <search condition> of *TC*.

b) REFERENCES privilege on every column identified by a <column reference> contained in the applicable <search condition> of *TC*.

c) USAGE privilege on every domain, every collation, every character set, and every transliteration whose names are contained in any applicable <search condition> of *TC*.

d) USAGE privilege on any user-defined type *UDT* such that some <data type> contained in the applicable <search condition> of *TC* is usage-dependent on *UDT*.

e) EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any applicable <search condition> of *TC*.

f) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in any applicable <search condition> of *TC* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

g) SELECT privilege on any column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in any applicable <search condition> of *TC*.

h) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of any <reference resolution> that is contained in any applicable <search condition> of *TC*.

i) SELECT privilege on the scoped table of any <reference resolution> that is contained in any applicable <search condition> of *TC*.

j) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in the applicable <search condition> of *TC*.

18) Let *AX* be any assertion descriptor included in *S1*. *AX* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

a) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in the applicable <search condition> of *AX*.

b) REFERENCES privilege on every column identified by a <column reference> contained in the applicable <search condition> of *AX*.

c) USAGE privilege on every domain, every collation, every character set, and every transliteration whose names are contained in any applicable <search condition> of *AX*.

d) USAGE privilege on any user-defined type *UDT* such that some <data type> contained in the applicable <search condition> of *AX* is usage-dependent on *UDT*.

e) EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any applicable <search condition> of *AX*.

f) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in the applicable <search condition> of *AX* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

g) SELECT privilege on any column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in any applicable <search condition> of *AX*.

h) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of any <reference resolution> that is contained in any applicable <search condition> of *AX*.

i) SELECT privilege on the scoped table of any <reference resolution> that is contained in any applicable <search condition> of *AX*.

j) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in the applicable <search condition> of *AX*.

19) Let *TR* be any trigger descriptor included in *S1*. *TR* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

a) TRIGGER privilege on the subject table of *TR*.

b) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in any <search condition> of *TR*.

c) SELECT privilege on every column identified by a <column reference> contained in any <search condition> of *TR*.

d) USAGE privilege on every domain, collation, character set, and transliteration whose name is contained in any <search condition> of *TR*.

e) USAGE privilege on any user-defined type *UDT* such that some <data type> contained in any <search condition> of *TR* is usage-dependent on *UDT*.

f) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in any <search condition> of *TR* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

g) EXECUTE privilege on the SQL-invoked routine that is the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any <search condition> of *TR*.

h) EXECUTE privilege on the SQL-invoked routine that is the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in the <triggered SQL statement> of *TR*.

i) SELECT privilege on at least one column of every table identified by a <table reference> contained in a <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

j) SELECT privilege on at least one column of every table identified by a <table reference> contained in a <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <triggered SQL statement> of *TR*.

k) SELECT privilege on at least one column of every table identified by a <table reference> and <column reference> contained in a <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

l) SELECT privilege on at least one column of every table identified by a <table reference> and <column reference> contained in a <value expression> simply contained in a an <update source> or an <assigned row>contained in the <triggered SQL statement> of *TR*.

m) INSERT privilege on every column

Case:

i) Identified by a <column name> contained in the <insert column list> of an <insert statement> or a <merge statement> contained in the <triggered SQL statement> of *TR*.

ii) Of the table identified by the <table name> immediately contained in an <insert statement> that does not contain an <insert column list> and that is contained in the <triggered SQL statement> of *TR*.

iii) Of the table identified by the <target table> contained in a <merge statement> that contains a <merge insert specification> and that does not contain an <insert column list> and that is contained in the <triggered SQL statement> of *TR*.

n) UPDATE privilege on every column identified by a <column name> is contained in an <object column> contained in either an <update statement: positioned>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

o) DELETE privilege on every table identified by a <table name> contained in either a <delete statement: positioned> or a <delete statement: searched> contained in the <triggered SQL statement> of *TR*.

p) USAGE privilege on every domain, collation, character set, transliteration, and sequence generator whose name is contained in the <triggered SQL statement> of *TR*.

q) USAGE privilege on any user-defined type *UDT* such that some <data type> contained in the <triggered SQL statement> of *TR* is usage-dependent on *UDT*.

r) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in any <triggered SQL statement> of *TR* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

s) SELECT privilege on any column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in any of the following:

   i) A <search condition> of *TR*.

   ii) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

   iii) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <triggered SQL statement> of *TR*.

   iv) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

   v) A <value expression> contained in an <update source> or an <assigned row> contained in the <triggered SQL statement> of *TR*.

t) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of any <reference resolution> that is contained in any of the following:

   i) A <search condition> of *TR*.

   ii) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

   iii) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <triggered SQL statement> of *TR*.

   iv) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

   v) A <value expression> contained in an <update source> or an <assigned row> contained in the <triggered SQL statement> of *TR*.

u) SELECT privilege on the scoped table of any <reference resolution> contained in any of the following:

   i) A <search condition> of *TR*.

   ii) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

iii) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <triggered SQL statement> of *TR*.

iv) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

v) A <value expression> contained in an <update source> or an <assigned row> contained in the <triggered SQL statement> of *TR*.

v) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in the <triggered SQL statement> of *TR*.

20) Let *DC* be any domain constraint descriptor included in *S1*. *DC* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

a) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in *TR*.

b) REFERENCES privilege on every column identified by a <column reference> contained in the <search condition> of *DC*.

c) USAGE privilege on every domain, every user-defined type, every collation, every character set, and every transliteration whose names are contained in any <search condition> of *DC*.

d) USAGE privilege on any user-defined type *UDT* such that some <data type> contained in any <search condition> of *DC* is usage-dependent on *UDT*.

e) EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any <search condition> of *DC*.

f) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in any <search condition> of *DC* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

g) SELECT privilege on any column identified by a <column reference> contained in a <scalar subquery> that is equivalent to some <dereference operation> contained in any <search condition> of *DC*.

h) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of any <reference resolution> that is contained in any <search condition> of *DC*.

i) SELECT privilege on the scoped table of any <reference resolution> that is contained in contained in any <search condition> of *DC*.

j) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in the <triggered SQL statement> of *TR*.

21) For every domain descriptor *DO* included in *S1*, *DO* is said to be *lost* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on every character set included in the data type descriptor included in *DO*.

22) For every table descriptor *TD* contained in *S1*, for every column descriptor *CD* included in *TD*, *CD* is said to be *lost* if any of the following are true:

a) The revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any character set included in the data type descriptor included in *CD*.

b) The revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any user-defined type *UDT* such that a data type descriptor included in *CD* describes a type that is usage-dependent on *UDT*.

c) The name of the domain *DN* included in *CD*, if any, identifies a lost domain descriptor and the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any character set included in the data type descriptor of the domain descriptor of *DN*.

23) For every SQL-client module *MO*, let *G* be the <module authorization identifier> that owns *MO*. *MO* is said to be *lost* if the revoke destruction action would result in *G* no longer having in its applicable privileges USAGE privilege on the character set referenced in the <module character set specification> of *MO*.

24) For every user-defined type descriptor *DT* included in *S1*, *DT* is said to be *abandoned* if any of the following are true:

a) The revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any user-defined type *UDT* such that a data type descriptor included in *DT* describes a type that is usage-dependent on *UDT*.

b) The revoke destruction action would result in *A1* no longer having in its applicable privileges the UNDER privilege on any user-defined type that is a direct supertype of *DT*.

25) *S1* is said to be *lost* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the default character set included in the *S1*.

26) For every collation descriptor *CN* contained in *S1*, *CN* is said to be *impacted* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the collation whose name is contained in the <existing collation name> of *CN*.

27) For every character set descriptor *CSD* contained in *S1*, *CSD* is said to be *impacted* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the collation whose name is contained in *CSD*.

28) For every descriptor included in *S1* that includes a data type descriptor *DTD*, *DTD* is said to be *impacted* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the collation whose name is included in *DTD*.

29) Let *RD* be any routine descriptor with an SQL security characteristic of DEFINER that is included in *S1*. *RD* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges all of the following:

a) EXECUTE privilege on the SQL-invoked routine that is the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in the <routine body> of *RD*.

b) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the SQL routine body of *RD*.

c) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <table expression> or <select list> immediately contained in a <select statement: single row> contained in the SQL routine body of *RD*.

d) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the SQL routine body of *RD*.

e) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <value expression> simply contained in an <update source> or an <assigned row> contained in the SQL routine body of *RD*.

f) SELECT privilege on at least one column identified by a <column reference> contained in a <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

g) SELECT privilege on at least one column identified by a <column reference> contained in a <value expression> simply contained in an <update source> or an <assigned row> contained in the SQL routine body of *RD*.

h) INSERT privilege on each column

   Case:

   i) Identified by a <column name> contained in the <insert column list> of an <insert statement> or a <merge statement> contained in the SQL routine body of *RD*.

   ii) Of the table identified by the <table name> immediately contained in an <insert statement> that does not contain an <insert column list> and that is contained in the SQL routine body of *RD*.

   iii) Of the table identified by the <target table> immediately contained in a <merge statement> that contains a <merge insert specification> and that does not contain an <insert column list> and that is contained in the SQL routine body of *RD*.

i) UPDATE privilege on each column whose name is contained in an <object column> contained in either an <update statement: positioned>, an <update statement: searched>, or a <merge statement> contained in the SQL routine body of *RD*.

j) DELETE privilege on each table whose name is contained in a <table name> contained in either a <delete statement: positioned> or a <delete statement: searched> contained in the SQL routine body of *RD*.

k) USAGE privilege on each domain, collation, character set, transliteration, and sequence generator whose name is contained in the SQL routine body of *RD*.

l) USAGE privilege on each user-defined type *UDT* such that a declared type of any SQL parameter, returns data type, or result cast included in *RD* is usage-dependent on *UDT*.

m) USAGE privilege on each user-defined type *UDT* such that some <data type> contained in the SQL routine body of *RD* is usage-dependent on *UDT*.

n) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in the SQL routine body of *RI* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

o) SELECT privilege on any column identified by a <column reference> contained in a <scalar subquery> that is equivalent to a <dereference operation> contained in any of the following:

   i) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <SQL routine body> of *RD*.

ii) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <SQL routine body> of *RD*.

iii) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

iv) A <value expression> contained in an <update source> or an <assigned row> contained in the <SQL routine body> of *RD*.

p) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of the scoped table of any <reference resolution> that is contained in any of the following:

i) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the SQL routine body of *RD*.

ii) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the SQL routine body of *RD*.

iii) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the SQL routine body of *RD*.

iv) A <value expression> simply contained in an <update source> or an <assigned row> contained in the SQL routine body of *RD*.

q) SELECT privilege on the scoped table of any <reference resolution> that is contained in any of the following:

i) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <SQL routine body> of *RD*.

ii) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <SQL routine body> of *RD*.

iii) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

iv) A <value expression> contained in an <update source> or an <assigned row> contained in the <SQL routine body> of *RD*.

r) SELECT privilege WITH HIERARCHY OPTION on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in the <SQL routine body> of *RD*.

30) For every table descriptor *TD* included in *S1*, for every column descriptor *CD* included in *TD*, *CD* is said to be *contaminated* if *CD* includes one of the following:

a) A user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

b) A reference type descriptor that includes a user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

c) A collection type descriptor that includes a user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

d) A collection type descriptor that includes a reference type descriptor that includes a user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

31) If RESTRICT is specified, and there exists an abandoned privilege descriptor, abandoned view, abandoned table constraint, abandoned assertion, abandoned domain constraint, lost domain, lost column, lost schema, or a descriptor that includes an impacted data type descriptor, impacted collation, impacted character set, abandoned user-defined type, forsaken column descriptor, forsaken domain descriptor, or abandoned routine descriptor, then an exception condition is raised: *dependent privilege descriptors still exist*.

32) If CASCADE is specified, then the impact on an SQL-client module that is determined to be a lost module is implementation-defined.

33) Case:

a) If the <revoke statement> is a <revoke privilege statement>, then

Case:

i) If neither WITH HIERARCHY OPTION nor GRANT OPTION FOR is specified, then:

1) All abandoned privilege descriptors are destroyed.

2) The identified privilege descriptors are destroyed.

3) The modified privilege descriptors are set to indicate that they are not grantable.

ii) If WITH HIERARCHY OPTION is specified, then the WITH HIERARCHY OPTION is removed from all identified and abandoned privilege descriptors, if present.

iii) If GRANT OPTION FOR is specified, then

Case:

1) If CASCADE is specified, then all abandoned privilege descriptors are destroyed.

2) Otherwise, if there are any privilege descriptors directly dependent on an identified privilege descriptor that are not modified privilege descriptors, then an exception condition is raised: *dependent privilege descriptors still exist*.

The identified privilege descriptors and the modified privilege descriptors are set to indicate that they are not grantable.

b) If the <revoke statement> is a <revoke role statement>, then:

i) If CASCADE is specified, then all abandoned role authorization descriptors are destroyed.

ii) All abandoned privilege descriptors are destroyed.

iii) Case:

1) If ADMIN OPTION FOR is specified, then the identified role authorization descriptors are set to indicate that they are not grantable.

2) If ADMIN OPTION FOR is not specified, then the identified role authorization descriptors are destroyed.

34) For every abandoned view descriptor *V*, let *S1.VN* be the <table name> of *V*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW S1.VN CASCADE
```

35) For every abandoned table descriptor *T*, let *S1*.*TN* be the <table name> of *T*. The following <drop table statement> is effectively executed without further Access Rule checking:

```
DROP TABLE S1.TN CASCADE
```

36) For every abandoned table constraint descriptor *TC*, let *S1*.*TCN* be the <constraint name> of *TC* and let *S2*.*T2* be the <table name> of the table that contains *TC* (*S1* and *S2* possibly equivalent). The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE S2.T2
    DROP CONSTRAINT S1.TCN
    CASCADE
```

37) For every abandoned assertion descriptor *AX*, let *S1*.*AXN* be the <constraint name> of *AX*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION S1.AXN CASCADE
```

38) For every abandoned trigger descriptor *TR*, let *S1*.*TRN* be the <trigger name> of *TR*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER S1.TRN
```

39) For every abandoned domain constraint descriptor *DC*, let *S1*.*DCN* be the <constraint name> of *DC* and let *S2*.*DN* be the <domain name> of the domain that contains *DC*. The following <alter domain statement> is effectively executed without further Access Rule checking:

```
ALTER DOMAIN S2.DN
    DROP CONSTRAINT S1.DCN
```

40) For every lost column descriptor *CD*, let *S1*.*TN* be the <table name> of the table whose descriptor includes the descriptor *CD* and let *CN* be the <column name> of *CD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE S1.TN
    DROP COLUMN CN CASCADE
```

41) For every lost domain descriptor *DO*, let *S1*.*DN* be the <domain name> of *DO*. The following <drop domain statement> is effectively executed without further Access Rule checking:

```
DROP DOMAIN S1.DN CASCADE
```

42) For every lost schema *S1*, the default character set of that schema is modified to include the name of the implementation-defined <character set specification> that would have been this schema's default character set had the <schema definition> not specified a <schema character set specification>.

43) If the object identified by *O* is a collation, let *OCN* be the name of that collation.

44) For every descriptor that includes an impacted data type descriptor *DTD*, *DTD* is modified such that it does not include *OCN*.

45) For every impacted collation descriptor *CD* with included collation name *CN*, the following <drop collation statement> is effectively executed without further Access Rule checking:

```
DROP COLLATION CN CASCADE
```

46) For every impacted character set descriptor *CSD* with included character set name *CSN*, *CSD* is modified so that the included collation name is the name of the default collation for the character set on which *CSD* is based.

47) For every abandoned user-defined type descriptor *DT* with <user-defined type name> *S1.DTN*, the following <drop data type statement> is effectively executed without further Access Rule checking:

```
DROP TYPE S1.DTN CASCADE
```

48) For every abandoned SQL-invoked routine descriptor *RD*, let *R* be the SQL-invoked routine whose descriptor is *RD*. Let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

49) If the <revoke statement> is a <revoke privilege statement>, then:

    a) For every combination of <grantee> and <action> on *O* specified in <privileges>, if there is no corresponding privilege descriptor in the set of identified privilege descriptors, then a completion condition is raised: *warning — privilege not revoked*.

    b) If ALL PRIVILEGES was specified, then for each <grantee>, if no privilege descriptors were identified, then a completion condition is raised: *warning — privilege not revoked*.

50) For every contaminated column descriptor *CD*, let *S1.TN* be the <table name> of the table whose descriptor includes the descriptor *CD* and let *CN* be the <column name> of *CD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE S1.TN
    DROP COLUMN CN CASCADE
```

Conformance Rules

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

2) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke statement> that contains a <drop behavior> that contains CASCADE.

3) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke option extension> that contains GRANT OPTION FOR.

4) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke statement> that contains a <privileges> that contains an <object name> where the owner of the SQL-schema that is specified explicitly or implicitly in the <object name> is not the current authorization identifier.

5) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke statement> such that there exists a privilege descriptor *PD* that satisfies all the following conditions:

a) *PD* identifies the object identified by <object name> simply contained in <privileges> contained in the <revoke statement>.

b) *PD* identifies the <grantee> identified by any <grantee> simply contained in <revoke statement> and that <grantee> does not identify the owner of the SQL-schema that is specified explicitly or implicitly in the <object name> simply contained in <privileges> contained in the <revoke statement>.

c) *PD* identifies the action identified by the <action> simply contained in <privileges> contained in the <revoke statement>.

d) *PD* indicates that the privilege is grantable.

6) Without Feature S081, “Subtables”, conforming SQL language shall not contain a <revoke option extension> that contains HIERARCHY OPTION FOR.

12.8 Grantor determination

Function

Determine the grantor of a privilege or role authorization, or the intended owner of a role.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *G* be the *GRANTOR* specified in an application of this Subclause.
2) The grantor *A* is derived from *G* as follows.

   Case:

   a) If *G* is OMITTED, then

      Case:

      i) If there is a current user identifier, then *A* is the current user identifier.

      ii) Otherwise, *A* is the current role name.

   b) If *G* is CURRENT_USER, then

      Case:

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

      ii) Otherwise, *A* is the current user identifier.

   c) If *G* is CURRENT_ROLE, then

      Case:

      i) If there is no current role name, then an exception condition is raised: *invalid grantor*.

      ii) Otherwise, *A* is the current role name.

Conformance Rules

1) Without Feature T332, “Extended roles”, conforming SQL language shall contain no <grantor>.

(Blank page)

13 SQL-client modules

This Clause is modified by Clause 11, "SQL-client modules", in ISO/IEC 9075-4.
This Clause is modified by Clause 14, "SQL-client modules", in ISO/IEC 9075-9.
This Clause is modified by Clause 13, "SQL-client modules", in ISO/IEC 9075-14.

13.1 <SQL-client module definition>

This Subclause is modified by Subclause 14.1, "<SQL-client module definition>", in ISO/IEC 9075-9.

Function

Define an SQL-client module.

Format

```
<SQL-client module definition> ::=
  <module name clause> <language clause> <module authorization clause>
      [ <module path specification> ]
      [ <module transform group specification> ]
      [ <module collations> ]
      [ <temporary table declaration>... ]
      <module contents>...

<module authorization clause> ::=
    SCHEMA <schema name>
  | AUTHORIZATION <module authorization identifier>
      [ FOR STATIC { ONLY | AND DYNAMIC } ]
  | SCHEMA <schema name> AUTHORIZATION <module authorization identifier>
      [ FOR STATIC { ONLY | AND DYNAMIC } ]

<module authorization identifier> ::=
  <authorization identifier>

<module path specification> ::=
  <path specification>

<module transform group specification> ::=
  <transform group specification>

<module collations> ::=
  <module collation specification>...

<module collation specification> ::=
  COLLATION <collation name> [ FOR <character set specification list> ]

<character set specification list> ::=
```

```
  <character set specification> [ { <comma> <character set specification> }... ]

<module contents> ::=
    <declare cursor>
  | <dynamic declare cursor>
  | <externally-invoked procedure>
```

Syntax Rules

1) The <language clause> shall not specify SQL.

2) If SCHEMA <schema name> is not specified, then a <schema name> equivalent to <module authorization identifier> is implicit.

3) If the explicit or implicit <schema name> does not specify a <catalog name>, then an implementation-defined <catalog name> is implicit.

4) The implicit or explicit <catalog name> is the implicit <catalog name> for all unqualified <schema name>s in the <SQL-client module definition>.

5) If <module path specification> is not specified, then a <module path specification> containing an implementation-defined <schema name list> that contains the <schema name> contained in <module authorization clause> is implicit.

6) The explicit or implicit <catalog name> of each <schema name> contained in the <schema name list> of the <module path specification> shall be equivalent to the <catalog name> of the explicit or implicit <schema name> contained in <module authorization clause>.

7) The <schema name list> of the explicit or implicit <module path specification> is used as the SQL-path of the <SQL-client module definition>. The SQL-path is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in the <SQL-client module definition>.

8) Case:

   a) If <module transform group specification> is not specified, then a <module transform group specification> containing a <multiple group specification> with a <group specification> *GS* for each <host parameter declaration> contained in <host parameter declaration list> of each <externally-invoked procedure> contained in <SQL-client module definition> whose <host parameter data type> *UDT* identifies a user-defined type with no <locator indication> is implicit. The <group name> of *GS* is implementation-defined and its <path-resolved user-defined type name> is *UDT*.

   b) If <module transform group specification> contains a <single group specification> with a <group name> *GN*, then a <module transform group specification> containing a <multiple group specification> that contains a <group specification> *GS* for each <host parameter declaration> contained in <host parameter declaration list> of each <externally-invoked procedure> contained in <SQL-client module definition> whose <host parameter data type> *UDT* identifies a user-defined type with no <locator indication> is implicit. The <group name> of *GS* is *GN* and its <path-resolved user-defined type name> is *UDT*.

   c) If <module transform group specification> contains a <multiple group specification> *MGS*, then a <module transform group specification> containing <multiple group specification> that contains *MGS* extended with a <group specification> *GS* for each <host parameter declaration> contained in <host parameter declaration list> of each <externally-invoked procedure> contained in <SQL-client module definition> whose <host parameter data type> *UDT* identifies a user-defined type with no <locator

indication> 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 *UDT*.

9) No two <character set specification>s contained in any <module collation specification> shall be equivalent.

10) A <module collation specification> *MCS* specifies the SQL-client module collation for one or more character sets for the SQL-client module. Let *CO* be the collation identified by the <collation name> contained in *MCS*.

Case:

a) If <character set specification list> is specified, then the collation specified by *CO* shall be applicable to every character set identified by a <character set specification> simply contained in the <module collation specification>. For each character set specified, the SQL-client module collation for that character set is set to *CO*.

b) Otherwise, the character sets for which the SQL-client module collation is set to *CO* are implementation-defined.

11) A <declare cursor> shall precede in the text of the <SQL-client module definition> any <externally-invoked procedure> that references the <cursor name> of the <declare cursor>.

12) A <dynamic declare cursor> shall precede in the text of the <SQL-client module definition> any <externally-invoked procedure> that references the <cursor name> of the <dynamic declare cursor>.

13) If neither FOR STATIC ONLY nor FOR STATIC AND DYNAMIC is specified, then FOR STATIC AND DYNAMIC is implicit.

14) For every <declare cursor> in an <SQL-client module definition>, the <SQL-client module definition> shall contain exactly one <open statement> that specifies the <cursor name> declared in the <declare cursor>.

NOTE 358 — See the Syntax Rules of Subclause 14.1, "<declare cursor>".

15) Let *EIP1* and *EIP2* be two <externally-invoked procedure>s contained in an <SQL-client module definition> that have the same number of <host parameter declaration>s and immediately contain a <fetch statement> referencing the same <cursor name>. Let *n* be the number of <host parameter declaration>s. Let $P1_i$, 1 (one) $\le i \le n$, be the *i*-th <host parameter declaration> of *EIP1*. Let $DT1_i$ be the <data type> contained in $P1_i$. Let $P2_i$ be the *i*-th <host parameter declaration> of *EIP2*. Let $DT2_i$ be the <data type> contained in $P2_i$. For each *i*, 1 (one) $\le i \le n$,

Case:

a) If $DT1_i$ and $DT2_i$ both identify a binary large object type, then either $P1_i$ and $P2_i$ shall both be binary large object locator parameters or neither shall be binary large object locator parameters.

b) If $DT1_i$ and $DT2_i$ both identify a character large object type, then either $P1_i$ and $P2_i$ shall both be character large object locator parameters or neither shall be character large object locator parameters.

c) If $DT1_i$ and $DT2_i$ both identify an array type, then either $P1_i$ and $P2_i$ shall both be array locator parameters or neither shall be array locator parameters.

d) If $DT1_i$ and $DT2_i$ both identify a multiset type, then either $P1_i$ and $P2_i$ shall both be multiset locator parameters or neither shall be multiset locator parameters.

e) If $DT1_i$ and $DT2_i$ both identify a user-defined type, then either $P1_i$ and $P2_i$ shall both be user-defined type locator parameters or neither shall be user-defined type locator parameters.

Access Rules

None.

General Rules

1) If the SQL-agent that performs a call of an <externally-invoked procedure> in an <SQL-client module definition> is not a program that conforms to the programming language standard for the programming language specified by the <language clause> of that <SQL-client module definition>, then the effect is implementation-dependent.

2) If the SQL-agent performs calls of <externally-invoked procedure>s from more than one Ada task, then the results are implementation-dependent.

3) If FOR STATIC ONLY is specified, then the SQL-client module includes an indication that prepared statements resulting from execution of externally-invoked procedures included in that module have no owner.

4) After the last time that an SQL-agent performs a call of an <externally-invoked procedure>:

   a) A <rollback statement> or a <commit statement> is effectively executed. If an unrecoverable error has occurred, or if the SQL-agent 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. If the implementation choice is <commit statement>, then all holdable cursors are first closed. The determination of whether an SQL-agent has terminated unexpectedly is implementation-dependent.

   b) For every SQL descriptor area that is currently allocated within an SQL-session associated with the SQL-agent, let *D* be the <descriptor name> of that SQL descriptor area; a <deallocate descriptor statement> that specifies

      ```
      DEALLOCATE DESCRIPTOR D
      ```

      is effectively executed.

   c) All SQL-sessions associated with the SQL-agent are terminated.

Conformance Rules

1) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <module path specification>.

2) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <module transform group specification>.

3) Without Feature F693, “SQL-session and client module collations”, conforming SQL language shall not contain a <module collation specification>.

4) Without Feature B051, “Enhanced execution rights”, conforming SQL language shall not contain a <module authorization clause> that immediately contains FOR STATIC ONLY or FOR STATIC AND DYNAMIC.

5) Without Feature B111, “Module language Ada”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains ADA.

6) Without Feature B112, “Module language C”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains C.

7) Without Feature B113, “Module language COBOL”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains COBOL.

8) Without Feature B114, “Module language Fortran”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains FORTRAN.

9) Without Feature B115, “Module language MUMPS”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains M.

10) Without Feature B116, “Module language Pascal”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains PASCAL.

11) Without Feature B117, “Module language PL/I”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains PLI.

13.2 <module name clause>

Function

Name an SQL-client module.

Format

```
<module name clause> ::=
  MODULE [ <SQL-client module name> ] [ <module character set specification> ]

<module character set specification> ::=
  NAMES ARE <character set specification>
```

Syntax Rules

1) If a <module name clause> does not specify an <SQL-client module name>, then the <SQL-client module definition> is unnamed.

2) The <SQL-client module name> shall not be equivalent to the <SQL-client module name> of any other <SQL-client module definition> in the same SQL-environment.

   NOTE 359 — An SQL-environment may have multiple <SQL-client module definition>s that are unnamed.

3) If the <language clause> of the containing <SQL-client module definition> specifies ADA, then an <SQL-client module name> shall be specified, and that <SQL-client module name> shall be a valid Ada library unit name.

4) If a <module character set specification> is not specified, then a <module character set specification> that specifies an implementation-defined character set that contains at least every character that is in <SQL language character> is implicit.

Access Rules

None.

General Rules

1) If <SQL-client module name> is specified, then, in the SQL-environment, the containing <SQL-client module definition> has the name given by <SQL-client module name>.

Conformance Rules

1) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <module character set specification>.

13.3 <externally-invoked procedure>

This Subclause is modified by Subclause 14.2, "<externally-invoked procedure>", in ISO/IEC 9075-9.

Function

Define an externally-invoked procedure.

Format

```
<externally-invoked procedure> ::=
  PROCEDURE <procedure name> <host parameter declaration list> <semicolon>
      <SQL procedure statement> <semicolon>

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

<host parameter declaration> ::=
    <host parameter name> <host parameter data type>
  | <status parameter>

<host parameter data type> ::=
  <data type> [ <locator indication> ]

<status parameter> ::=
  SQLSTATE
```

Syntax Rules

1) The <procedure name> shall not be equivalent to the <procedure name> of any other <externally-invoked procedure> in the containing <SQL-client module definition>.

   NOTE 360 — The <procedure name> should be a standard-conforming procedure, function, or routine name of the language specified by the subject <language clause>. Failure to observe this recommendation will have implementation-dependent effects.

2) The <host parameter name> of each <host parameter declaration> in an <externally-invoked procedure> shall not be equivalent to the <host parameter name> of any other <host parameter declaration> in that <externally-invoked procedure>.

3) Any <host parameter name> contained in the <SQL procedure statement> of an <externally-invoked procedure> shall be specified in a <host parameter declaration> in that <externally-invoked procedure>.

4) If <locator indication> is simply contained in <host parameter declaration>, then:

   a) The declared type *T* identified by the <data type> immediately contained in <host parameter data type> shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

   b) If *T* is a binary large object type, then the host parameter identified by <host parameter name> is called a *binary large object locator parameter*.

c) If *T* is a character large object type, then the host parameter identified by <host parameter name> is called a *character large object locator parameter*.

d) If *T* is an array type, then the host parameter identified by <host parameter name> is called an *array locator parameter*.

e) If *T* is a multiset type, then the host parameter identified by <host parameter name> is called a *multiset locator parameter*.

f) If *T* is a user-defined type, then the host parameter identified by <host parameter name> is called a *user-defined type locator parameter*.

5) A call of an <externally-invoked procedure> shall supply *n* arguments, where *n* is the number of <host parameter declaration>s in the <externally-invoked procedure>.

6) An <externally-invoked procedure> shall contain one <status parameter> referred to as an *SQLSTATE host parameter*. The SQLSTATE host parameter is referred to as a *status parameter*.

7) The Syntax Rules of Subclause 9.6, "Host parameter mode determination", with <host parameter declaration> as *PD* and <SQL procedure statement> as *SPS* for each <host parameter declaration>, are applied to determine whether the corresponding host parameter is an input host parameter, an output host parameter, or both an input host parameter and an output host parameter.

8) The Syntax Rules of Subclause 13.4, "Calls to an <externally-invoked procedure>", shall be satisfied.

Access Rules

None.

General Rules

1) An <externally-invoked procedure> defines an *externally-invoked procedure* that may be called by an SQL-agent.

2) If the <SQL-client module definition> that contains the <externally-invoked procedure> is associated with an SQL-agent that is associated with another <SQL-client module definition> that contains an <externally-invoked procedure> with equivalent <procedure name>s, then the effect is implementation-defined.

3) The language identified by the <language name> contained in the <language clause> of the <SQL-client module definition> that contains an <externally-invoked procedure> is the *caller language* of the <externally-invoked procedure>.

4) If the SQL-agent that performs a call of a <externally-invoked procedure> is not a program that conforms to the programming language standard specified by the caller language of the <externally-invoked procedure>, then the effect is implementation-dependent.

5) If the caller language of an <externally-invoked procedure> is ADA and the SQL-agent performs calls of <externally-invoked procedure>s from more than one Ada task, then the results are implementation-dependent.

6) If the <SQL-client module definition> that contains the <externally-invoked procedure> has an explicit <module authorization identifier> *MAI* that is not equivalent to the SQL-session <authorization identifier> *SAI*, then:

a) Whether or not *SAI* can invoke <externally-invoked procedure>s in an <SQL-client module definition> with explicit <module authorization identifier> *MAI* is implementation-defined, as are any restrictions pertaining to such invocation.

b) If *SAI* is restricted from invoking an <externally-invoked procedure> in an <SQL-client module definition> with explicit <module authorization identifier> *MAI*, then an exception condition is raised: *invalid authorization specification*.

7) If the value of any input host parameter provided by the SQL-agent falls outside the set of allowed values of the declared type of the host parameter, or if the value of any output host parameter resulting from the execution of the <externally-invoked procedure> falls outside the set of values supported by the SQL-agent for that host parameter, then the effect is implementation-defined. If the implementation-defined effect is the raising of an exception condition, then an exception condition is raised: *data exception — invalid parameter value*.

8) A copy of the top cell of the authorization stack is pushed onto the authorization stack. If the SQL-client module *M* that includes the externally-invoked procedure has an owner, then the top cell of the authorization stack is set to contain only the authorization identifier of the owner of *M*.

9) If this is the first invocation of any externally-invoked procedure of *M* in the current SQL-session, then for each cursor declaration descriptor *CDD* of a declared cursor of *M*, let *CID* be a new cursor instance descriptor whose cursor declaration descriptor is *CDD*, whose SQL-session is the current SQL-session, and whose state is closed.

10) Let *S* be the <SQL procedure statement> of the <externally-invoked procedure>.

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

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

Conformance Rules

1) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <host parameter data type> that simply contains a <data type> that specifies a structured type and that contains a <locator indication>.

2) Without Feature S232, "Array locators", conforming SQL language shall not contain a <host parameter data type> that simply contains an <array type> and that contains a <locator indication>.

3) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <host parameter data type> that simply contains a <multiset type> and that contains a <locator indication>.

13.4 Calls to an <externally-invoked procedure>

This Subclause is modified by Subclause 11.1, "Calls to an <externally-invoked procedure>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 14.3, "Calls to an <externally-invoked procedure>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 13.1, "Calls to an <externally-invoked procedure>", in ISO/IEC 9075-14.

Function

Define the call to an <externally-invoked procedure> by an SQL-agent.

Syntax Rules

1) Let *n* be the number of <host parameter declaration>s in the <externally-invoked procedure> *EP* being called. Let PD_i, 1 (one) ≤ *i* ≤ *n*, be the *i*-th <host parameter declaration>. Let PDT_i be the <data type> contained in PD_i.

2) If the caller language of the <externally-invoked procedure> is ADA, then:

   a) The SQL-implementation shall generate the source code of an Ada library unit package *ALUP* the name of which shall be

      Case:

      i) If the <SQL-client module name> *SCMN* of the <SQL-client module definition> <SQL-client module name> is a valid Ada identifier, then equivalent to *SCMN*.

      ii) Otherwise, implementation-defined.

   b) For each <externally-invoked procedure> of the <SQL-client module definition>, there shall appear within *ALUP* a subprogram declaration declaring a procedure.

      i) If <procedure name> is a valid Ada identifier, then the name of that procedure *PN* shall be equivalent to <procedure name>; otherwise, *PN* shall be implementation-defined.

      ii) The parameters in each Ada procedure declaration *APD* shall appear in the same order as the <host parameter declaration>s of the corresponding <externally-invoked procedure> *EIP*. If the names of the parameters declared in the <host parameter declaration>s of *EIP* are valid Ada identifiers, then the parameters in *APD* shall have parameter names that are equivalent to the names of the corresponding parameters declared in the <host parameter declaration>s contained in *EIP*; otherwise, the parameters in *APD* shall parameter names that are implementation-defined

      iii) The parameter modes and subtype marks used in the parameter specifications are constrained by the remaining paragraphs of this Subclause.

   c) For each *i*, 1 (one) ≤ *i* ≤ *n*, PDT_i shall not identify a data type listed in the "SQL data type" column of Table 16, "Data type correspondences for Ada", for which the corresponding row in the "Ada data type" column is 'None'.

d) The types of parameter specifications within the Ada subprogram declarations shall be taken from the library unit package `Interfaces.SQL` and its children `Numerics` and `Varying` and optional children `Adacsn` and `Adacsn.Varying`.

e) The declaration of the library unit package `Interfaces.SQL` shall conform to the following template:

```
package Interfaces.SQL is
– The declarations of CHAR and NCHAR may be subtype declarations
  type CHAR is (See the Syntax Rules)
  type NCHAR is (See the Syntax Rules)
  type SMALLINT is range bs .. ts;
  type INT is range bi .. ti;
  type BIGINT is range bb .. tb;
  type REAL is digits dr;
  type DOUBLE_PRECISION is digits dd;
  type BOOLEAN is new Boolean;
  subtype INDICATOR_TYPE is t;
  type SQLSTATE_TYPE is new CHAR (1 .. 5);
  package SQLSTATE_CODES is
    AMBIGUOUS_CURSOR_NAME_NO_SUBCLASS:
      constant SQLSTATE_TYPE :="3C000";
    ATTEMPT_TO_ASSIGN_TO_NON_UPDATABLE_COLUMN_NO_SUBCLASS:
      constant SQLSTATE_TYPE := "0U000";
    ATTEMPT_TO_ASSIGN_TO_ORDERING_COLUMN_NO_SUBCLASS:
      constant SQLSTATE_TYPE := "0V000";
    CARDINALITY_VIOLATION_NO_SUBCLASS:
      constant SQLSTATE_TYPE :="21000";
    CLI_SPECIFIC_CONDITION_NO_SUBCLASS:
      constant SQLSTATE_TYPE :="HY000";
    CONNECTION_EXCEPTION_NO_SUBCLASS:
      constant SQLSTATE_TYPE :="08000";
    CONNECTION_EXCEPTION_CONNECTION_DOES_NOT_EXIST:
      constant SQLSTATE_TYPE :="08003";
    CONNECTION_EXCEPTION_CONNECTION_FAILURE:
      constant SQLSTATE_TYPE :="08006";
    CONNECTION_EXCEPTION_CONNECTION_NAME_IN_USE:
      constant SQLSTATE_TYPE :="08002";
    CONNECTION_EXCEPTION_SQLCLIENT_UNABLE_TO_ESTABLISH_SQLCONNECTION:
      constant SQLSTATE_TYPE :="08001";
    CONNECTION_EXCEPTION_SQLSERVER_REJECTED_ESTABLISHMENT_OF_SQLCONNECTION:
      constant SQLSTATE_TYPE :="08004";
    CONNECTION_EXCEPTION_TRANSACTION_RESOLUTION_UNKNOWN:
      constant SQLSTATE_TYPE :="08007";
    DATA_EXCEPTION_NO_SUBCLASS:
      constant SQLSTATE_TYPE :="22000";
    DATA_EXCEPTION_ARRAY_ELEMENT_ERROR:
      constant SQLSTATE_TYPE :="2202E";


 DATA_EXCEPTION_ATTEMPT_TO_REPLACE_A_ZERO_LENGTH_STRING:
   constant SQLSTATE_TYPE :="2201U";

    DATA_EXCEPTION_CHARACTER_NOT_IN_REPERTOIRE:
      constant SQLSTATE_TYPE :="22021";
    DATA_EXCEPTION_DATETIME_FIELD_OVERFLOW:
      constant SQLSTATE_TYPE :="22008";
    DATA_EXCEPTION_DIVISION_BY_ZERO:
```

```
      constant SQLSTATE_TYPE :="22012";
    DATA_EXCEPTION_ERROR_IN_ASSIGNMENT:
      constant SQLSTATE_TYPE :="22005";
    DATA_EXCEPTION_ESCAPE_CHARACTER_CONFLICT:
      constant SQLSTATE_TYPE :="2200B";
    DATA_EXCEPTION_INDICATOR_OVERFLOW:
      constant SQLSTATE_TYPE :="22022";
    DATA_EXCEPTION_INTERVAL_FIELD_OVERFLOW:
      constant SQLSTATE_TYPE :="22015";
    DATA_EXCEPTION_INTERVAL_VALUE_OUT_OF_RANGE:
      constant SQLSTATE_TYPE :="2200P";
    DATA_EXCEPTION_INVALID_ARGUMENT_FOR_NATURAL_LOGARITHM:
      constant SQLSTATE_TYPE :="2201E";
    DATA_EXCEPTION_INVALID_ARGUMENT_FOR_POWER_FUNCTION:
      constant SQLSTATE_TYPE :="2201F";
    DATA_EXCEPTION_INVALID_ARGUMENT_FOR_WIDTH_BUCKET_FUNCTION:
      constant SQLSTATE_TYPE :="2201G";
    DATA_EXCEPTION_INVALID_CHARACTER_VALUE_FOR_CAST:
      constant SQLSTATE_TYPE :="22018";
    DATA_EXCEPTION_INVALID_DATETIME_FORMAT:
      constant SQLSTATE_TYPE :="22007";
    DATA_EXCEPTION_INVALID_ESCAPE_CHARACTER:
      constant SQLSTATE_TYPE :="22019";
    DATA_EXCEPTION_INVALID_ESCAPE_OCTET:
      constant SQLSTATE_TYPE :="2200D";
    DATA_EXCEPTION_INVALID_ESCAPE_SEQUENCE:
      constant SQLSTATE_TYPE :="22025";
    DATA_EXCEPTION_INVALID_INDICATOR_PARAMETER_VALUE:
      constant SQLSTATE_TYPE :="22010";
    DATA_EXCEPTION_INVALID_INTERVAL_FORMAT:
      constant SQLSTATE_TYPE :="22006";
    DATA_EXCEPTION_INVALID_PARAMETER_VALUE:
      constant SQLSTATE_TYPE :="22023";
    DATA_EXCEPTION_INVALID_PRECEDING_OR_FOLLOWING_SIZE_IN_WINDOW_FUNCTION:
      constant SQLSTATE_TYPE :="22013";
    DATA_EXCEPTION_INVALID_REGULAR_EXPRESSION:
      constant SQLSTATE_TYPE :="2201B";
    DATA_EXCEPTION_INVALID_REPEAT_ARGUMENT_IN_A_SAMPLE_CLAUSE:
      constant SQLSTATE_TYPE :="2202G";
    DATA_EXCEPTION_INVALID_SAMPLE_SIZE:
      constant SQLSTATE_TYPE :="2202H";
    DATA_EXCEPTION_INVALID_TIME_ZONE_DISPLACEMENT_VALUE:
      constant SQLSTATE_TYPE :="22009";
    DATA_EXCEPTION_INVALID_USE_OF_ESCAPE_CHARACTER:
      constant SQLSTATE_TYPE :="2200C";


  DATA_EXCEPTION_INVALID_XQUERY_OPTION_FLAG:
    constant SQLSTATE_TYPE :="2201T";

  DATA_EXCEPTION_INVALID_XQUERY_REGULAR_EXPRESSION:
    constant SQLSTATE_TYPE :="2201S";

  DATA_EXCEPTION_INVALID_XQUERY_REPLACEMENT_STRING:
    constant SQLSTATE_TYPE :="2201V";

     DATA_EXCEPTION_NULL_VALUE_NO_INDICATOR_PARAMETER:
```

```
  constant SQLSTATE_TYPE :="2200G";
DATA_EXCEPTION_MOST_SPECIFIC_TYPE_MISMATCH:
  constant SQLSTATE_TYPE :="22002";
DATA_EXCEPTION_MULTISET_VALUE_OVERFLOW:
  constant SQLSTATE_TYPE :="2200Q";
DATA_EXCEPTION_NONCHARACTER_IN_UCS_STRING:
  constant SQLSTATE_TYPE :="22029";
DATA_EXCEPTION_NULL_VALUE_NOT_ALLOWED:
  constant SQLSTATE_TYPE :="22004";
DATA_EXCEPTION_NULL_VALUE_SUBSTITUTED_FOR_MUTATOR_SUBJECT_PARAMETER:
  constant SQLSTATE_TYPE :="2202D";
DATA_EXCEPTION_NUMERIC_VALUE_OUT_OF_RANGE:
  constant SQLSTATE_TYPE :="22003";
DATA_EXCEPTION_SEQUENCE_GENERATOR_LIMIT_EXCEEDED:
  constant SQLSTATE_TYPE :="2200H";
DATA_EXCEPTION_STRING_DATA_LENGTH_MISMATCH:
  constant SQLSTATE_TYPE :="22026";
DATA_EXCEPTION_STRING_DATA_RIGHT_TRUNCATION:
  constant SQLSTATE_TYPE :="22001";
DATA_EXCEPTION_SUBSTRING_ERROR:
  constant SQLSTATE_TYPE :="22011";
DATA_EXCEPTION_TRIM_ERROR:
  constant SQLSTATE_TYPE :="22027";
DATA_EXCEPTION_UNTERMINATED_C_STRING:
  constant SQLSTATE_TYPE :="22024";
DATA_EXCEPTION_ZERO_LENGTH_CHARACTER_STRING:
  constant SQLSTATE_TYPE :="2200F";
DEPENDENT_PRIVILEGE_DESCRIPTORS_STILL_EXIST_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="2B000";
DIAGNOSTICS_EXCEPTION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="0Z000";
DIAGNOSTICS_EXCEPTION_MAXIMUM_NUMBER_OF_DIAGNOSTICS_AREAS_EXCEEDED:
  constant SQLSTATE_TYPE :="0Z001";
DYNAMIC_SQL_ERROR_NO_SUBCLASS:
  constant SQLSTATE_TYPE := "07000";
DYNAMIC_SQL_ERROR_CURSOR_SPECIFICATION_CANNOT_BE_EXECUTED:
  constant SQLSTATE_TYPE := "07003";
DYNAMIC_SQL_ERROR_INVALID_DATETIME_INTERVAL_CODE:
  constant SQLSTATE_TYPE := "0700F";
DYNAMIC_SQL_ERROR_INVALID_DESCRIPTOR_COUNT:
  constant SQLSTATE_TYPE := "07008";
DYNAMIC_SQL_ERROR_INVALID_DESCRIPTOR_INDEX:
  constant SQLSTATE_TYPE := "07009";
DYNAMIC_SQL_ERROR_PREPARED_STATEMENT_NOT_A_CURSOR_SPECIFICATION:
  constant SQLSTATE_TYPE := "07005";
DYNAMIC_SQL_ERROR_RESTRICTED_DATA_TYPE_ATTRIBUTE_VIOLATION:
  constant SQLSTATE_TYPE := "07006";
DYNAMIC_SQL_ERROR_DATA_TYPE_TRANSFORM_FUNCTION_VIOLATION:
  constant SQLSTATE_TYPE := "0700B";
DYNAMIC_SQL_ERROR_INVALID_DATA_TARGET:
  constant SQLSTATE_TYPE := "0700D";
DYNAMIC_SQL_ERROR_INVALID_LEVEL_VALUE:
  constant SQLSTATE_TYPE := "0700E";
DYNAMIC_SQL_ERROR_UNDEFINED_DATA_VALUE:
  constant SQLSTATE_TYPE := "0700C";
DYNAMIC_SQL_ERROR_USING_CLAUSE_DOES_NOT_MATCH_DYNAMIC_PARAMETER_SPEC:
  constant SQLSTATE_TYPE := "07001";
```

```
DYNAMIC_SQL_ERROR_USING_CLAUSE_DOES_NOT_MATCH_TARGET_SPEC:
  constant SQLSTATE_TYPE := "07002";
DYNAMIC_SQL_ERROR_USING_CLAUSE_REQUIRED_FOR_DYNAMIC_PARAMETERS:
  constant SQLSTATE_TYPE := "07004";
DYNAMIC_SQL_ERROR_USING_CLAUSE_REQUIRED_FOR_RESULT_FIELDS:
  constant SQLSTATE_TYPE := "07007";
EXTERNAL_ROUTINE_EXCEPTION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="38000";
EXTERNAL_ROUTINE_EXCEPTION_CONTAINING_SQL_NOT_PERMITTED:
  constant SQLSTATE_TYPE :="38001";
EXTERNAL_ROUTINE_EXCEPTION_MODIFYING_SQL_DATA_NOT_PERMITTED:
  constant SQLSTATE_TYPE :="38002";
EXTERNAL_ROUTINE_EXCEPTION_PROHIBITED_SQL_STATEMENT_ATTEMPTED:
  constant SQLSTATE_TYPE :="38003";
EXTERNAL_ROUTINE_EXCEPTION_READING_SQL_DATA_NOT_PERMITTED:
  constant SQLSTATE_TYPE :="38004";
EXTERNAL_ROUTINE_INVOCATION_EXCEPTION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="39000";
EXTERNAL_ROUTINE_INVOCATION_EXCEPTION_NULL_VALUE_NOT_ALLOWED:
  constant SQLSTATE_TYPE :="39004";
FEATURE_NOT_SUPPORTED_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="0A000";
FEATURE_NOT_SUPPORTED_MULTIPLE_ENVIRONMENT_TRANSACTIONS:
  constant SQLSTATE_TYPE :="0A001";
INTEGRITY_CONSTRAINT_VIOLATION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="23000";
INTEGRITY_CONSTRAINT_VIOLATION_RESTRICT_VIOLATION:
  constant SQLSTATE_TYPE :="23001";
INVALID_AUTHORIZATION_SPECIFICATION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="28000";
INVALID_CATALOG_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="3D000";
INVALID_CHARACTER_SET_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="2C000";
INVALID_COLLATION_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="2H000";
INVALID_CONDITION_NUMBER_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="35000";
INVALID_CONNECTION_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="2E000";
INVALID_CURSOR_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="34000";
INVALID_CURSOR_STATE_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="24000";
INVALID_GRANTOR_STATE_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="0L000";
INVALID_ROLE_SPECIFICATION:
  constant SQLSTATE_TYPE :="0P000";
INVALID_SCHEMA_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="3F000";
INVALID_SCHEMA_NAME_LIST_SPECIFICATION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="0E000";
INVALID_SQL_DESCRIPTOR_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="33000";
INVALID_SQL_INVOKED_PROCEDURE_REFERENCE_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="0M000";
INVALID_SQL_STATEMENT:
```

```
  constant SQLSTATE_TYPE :="30000";
INVALID_SQL_STATEMENT_IDENTIFIER_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="30000";
INVALID_SQL_STATEMENT_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="26000";
INVALID_TRANSFORM_GROUP_NAME_SPECIFICATION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="0S000";
INVALID_TRANSACTION_STATE_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="25000";
INVALID_TRANSACTION_STATE_ACTIVE_SQL_TRANSACTION:
  constant SQLSTATE_TYPE :="25001";
INVALID_TRANSACTION_STATE_BRANCH_TRANSACTION_ALREADY_ACTIVE:
  constant SQLSTATE_TYPE :="25002";
INVALID_TRANSACTION_STATE_HELD_CURSOR_REQUIRES_SAME_ISOLATION_LEVEL:
  constant SQLSTATE_TYPE :="25008";
INVALID_TRANSACTION_STATE_INAPPROPRIATE_ACCESS_MODE_FOR_BRANCH_TRANSACTION:
  constant SQLSTATE_TYPE :="25003";
INVALID_TRANSACTION_STATE_INAPPROPRIATE_ISOLATION_LEVEL_FOR_BRANCH_TRANSACTION:

  constant SQLSTATE_TYPE :="25004";
INVALID_TRANSACTION_STATE_NO_ACTIVE_SQL_TRANSACTION_FOR_BRANCH_TRANSACTION:
  constant SQLSTATE_TYPE :="25005";
INVALID_TRANSACTION_STATE_READ_ONLY_SQL_TRANSACTION:
  constant SQLSTATE_TYPE :="25006";
INVALID_TRANSACTION_STATE_SCHEMA_AND_DATA_STATEMENT_MIXING_NOT_SUPPORTED:
  constant SQLSTATE_TYPE :="25007";
INVALID_TRANSACTION_TERMINATION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="2D000";
LOCATOR_EXCEPTION_INVALID_SPECIFICATION:
  constant SQLSTATE_TYPE :="0F001";
LOCATOR_EXCEPTION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="0F000";
NO_DATA_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="02000";
NO_DATA_NO_ADDITIONAL_DYNAMIC_RESULT_SETS_RETURNED:
  constant SQLSTATE_TYPE :="02001";
NO_DATA_NO_DYNAMIC_RESULT_SETS_RETURNED:
  constant SQLSTATE_TYPE :="02001";
REMOTE_DATABASE_ACCESS_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="HZ000";
SAVEPOINT_EXCEPTION_INVALID_SPECIFICATION:
  constant SQLSTATE_TYPE :="3B001";
SAVEPOINT_EXCEPTION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="3B000";
SAVEPOINT_EXCEPTION_TOO_MANY:
  constant SQLSTATE_TYPE :="3B002";
SQL_ROUTINE_EXCEPTION_NO_SUBCLASS:
  constant SQLSTATE_TYPE :="2F000";
SQL_ROUTINE_EXCEPTION_FUNCTION_EXECUTED_NO_RETURN_STATEMENT:
  constant SQLSTATE_TYPE :="2F005";
SQL_ROUTINE_EXCEPTION_MODIFYING_SQL_DATA_NOT_PERMITTED:
  constant SQLSTATE_TYPE :="2F002";
SQL_ROUTINE_EXCEPTION_PROHIBITED_SQL_STATEMENT_ATTEMPTED:
  constant SQLSTATE_TYPE :="2F003";
SQL_ROUTINE_EXCEPTION_READING_SQL_DATA_NOT_PERMITTED:
  constant SQLSTATE_TYPE :="2F004";
SUCCESSFUL_COMPLETION_NO_SUBCLASS:
```

```
        constant SQLSTATE_TYPE :="00000";
      SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="42000";
      SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION_IN_DIRECT_STATEMENT_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="2A000";
      SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION_IN_DYNAMIC_STATEMENT_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="37000";
      TARGET_TABLE_DISAGREES_WITH_CURSOR_SPECIFICATION_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="0T000";
      TRANSACTION_ROLLBACK_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="40000";
      TRANSACTION_ROLLBACK_INTEGRITY_CONSTRAINT_VIOLATION:
        constant SQLSTATE_TYPE :="40002";
      TRANSACTION_ROLLBACK_SERIALIZATION_FAILURE:
        constant SQLSTATE_TYPE :="40001";
      TRANSACTION_ROLLBACK_STATEMENT_COMPLETION_UNKNOWN:
        constant SQLSTATE_TYPE :="40003";
      TRIGGERED_DATA_CHANGE_VIOLATION_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="27000";
      WARNING_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="01000";
      WARNING_ADDITIONAL_RESULT_SETS_RETURNED:
        constant SQLSTATE_TYPE :="0100D";
      WARNING_ARRAY_DATA_RIGHT_TRUNCATION:
        constant SQLSTATE_TYPE :="0102F";
      WARNING_ATTEMPT_TO_RETURN_TOO_MANY_RESULT_SETS:
        constant SQLSTATE_TYPE :="0100E";
      WARNING_CURSOR_OPERATION_CONFLICT:
        constant SQLSTATE_TYPE :="01001";
      WARNING_DEFAULT_VALUE_TOO_LONG_FOR_INFORMATION_SCHEMA:
        constant SQLSTATE_TYPE :="0100B";
      WARNING_DISCONNECT_ERROR:
        constant SQLSTATE_TYPE :="01002";
      WARNING_DYNAMIC_RESULT_SETS_RETURNED:
        constant SQLSTATE_TYPE :="0100C";
      WARNING_INSUFFICIENT_ITEM_DESCRIPTOR_AREAS:
        constant SQLSTATE_TYPE :="01005";
      WARNING_NULL_VALUE_ELIMINATED_IN_SET_FUNCTION:
        constant SQLSTATE_TYPE :="01003";
      WARNING_PRIVILEGE_NOT_GRANTED:
        constant SQLSTATE_TYPE :="01007";
      WARNING_PRIVILEGE_NOT_REVOKED:
        constant SQLSTATE_TYPE :="01006";
      WARNING_QUERY_EXPRESSION_TOO_LONG_FOR_INFORMATION_SCHEMA:
        constant SQLSTATE_TYPE :="0100A";
      WARNING_RESULT_SETS_RETURNED:
        constant SQLSTATE_TYPE :="0100C";
      WARNING_SEARCH_CONDITION_TOO_LONG_FOR_INFORMATION_SCHEMA:
        constant SQLSTATE_TYPE :="01009";
      WARNING_STATEMENT_TOO_LONG_FOR_INFORMATION_SCHEMA:
        constant SQLSTATE_TYPE :="0100F";
      WARNING_STRING_DATA_RIGHT_TRUNCATION_WARNING:
        constant SQLSTATE_TYPE :="01004";
      WITH_CHECK_OPTION_VIOLATION_NO_SUBCLASS:
        constant SQLSTATE_TYPE :="44000";
    end SQLSTATE_CODES;
  end Interfaces.SQL;
```

where *bs*, *ts*, *bi*, *ti*, *bb*, *tb*, *dr*, *dd*, *bsc*, and *tsc* are implementation-defined integer values. *t* is INT or SMALLINT, corresponding with an implementation-defined <exact numeric type> of indicator parameters.

NOTE 361 — The Ada identifier INVALID_SQL_STATEMENT appears for compatibility with earlier editions of ISO/IEC 9075. However, the intended symbol is INVALID_SQL_STATEMENT_IDENTIFIER_NO_SUBCLASS, which was added to correspond correctly with the exception condition name.

f) The library unit package `Interfaces.SQL.Numerics` shall contain a sequence of decimal fixed point type declarations of the following form.

```
type Scale_s is delta 10.0 ** - s digits max_p;
```

where *s* is an integer ranging from 0 (zero) to an implementation-defined maximum value and *max_p* is an implementation-defined integer maximum precision.

g) The library unit package `Interfaces.SQL.Varying` shall contain type or subtype declarations with the defining identifiers CHAR and NCHAR.

h) Let *SQLcsn* be a <character set name> and let *Adacsn* be the result of replacing <period>'s in *SQLcsn* with <underscore>s. If *Adacsn* is a valid Ada identifier, then the library unit packages `Interfaces.SQL.`*`Adacsn`* and `Interfaces.SQL.`*`Adacsn`*`.Varying` shall contain a type or subtype declaration with defining identifier CHAR. If *Adacsn* is not a valid Ada identifier, then the names of these packages shall be implementation-defined.

i) `Interfaces.SQL` and its children may contain context clauses and representation items as needed. These packages may also contain declarations of Ada character types as needed to support the declarations of the types CHAR and NCHAR.

NOTE 362 — If the implementation-defined character set specification used by default with a fixed-length character string type is Latin1, then the declaration

```
subtype CHAR is String;
```

within `Interfaces.SQL` and the declaration

```
subtype CHAR is
    Ada.Strings.Unbounded.Unbounded_String;
```

within `Interfaces.SQL.Varying` (assuming the appropriate context clause) conform to the requirements of this paragraph of this Subclause. If the character set underlying NATIONAL CHARACTER is supported by an Ada package specification `Host_Char_Pkg` that declares a type `String_Type` that stores strings over the given character set, and furthermore the package specification `Host_Char_Pkg_Varying` (not necessary distinct from `Host_Char_Pkg`) declares a type `String_Type_Varying` that reproduces the functionality of `Ada.Strings.Unbounded.Unbounded_String` over the national character string type (rather than Latin1), then the declaration

```
subtype NCHAR is Host_Char_Pkg.String_Type;
```

within `Interfaces.SQL` and the declaration

```
subtype NCHAR is Host_Char_Pkg_Varying.String_Type_Varying;
```

within `Interfaces.SQL.Varying` conform to the requirements of this paragraph. Similar comments apply to other character sets and the packages `Interfaces.SQL.`*`Adacsn`* and `Interfaces.SQL.`*`Adacsn`*`.Varying`.

j) The library unit package `Interfaces.SQL` shall contain declarations of the following form:

```
package CHARACTER_SET renames Interfaces.SQL.Adacsn;
subtype CHARACTER_TYPE is CHARACTER_SET.cst;
```

where *cst* is a data type capable of storing a single character from the default character set. The package `Interfaces.SQL.`*`Adacsn`* shall contain the necessary declaration for *cst*.

NOTE 363 — If the default character set is Latin1, then a declaration of the form:

```
package CHARACTER_SET is
   subtype cst is Character;
end CHARACTER_SET;
```

may be substituted for the renaming declaration of CHARACTER_SET.

k) The base type of the SQLSTATE parameter shall be `Interfaces.SQL.SQLSTATE_TYPE`.

l) The Ada parameter mode of the SQLSTATE parameter is out.

m) If the *i*-th <host parameter declaration> specifies a <data type> that is:

i) CHARACTER(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.CHAR`.

ii) CHARACTER VARYING(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.VARYING.CHAR`.

iii) NATIONAL CHARACTER(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.NCHAR`.

iv) NATIONAL CHARACTER VARYING(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.VARYING.NCHAR`.

v) CHARACTER(*L*) CHARACTER SET *csn* for some *L* and some character set name *csn*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.`*`Adacsn`*`.CHAR`.

vi) CHARACTER VARYING(*L*) CHARACTER SET *csn* for some *L* and some character set name *csn*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.`*`Adacsn`*`.VARYING.CHAR`.

If *P* is an actual parameter associated with the *i*-th parameter in a call to the encompassing procedure, then *P* shall be sufficient to hold a character string of length *L* in the appropriate character set.

NOTE 364 — If a character set uses fixed length encodings then the definition of the subtype CHAR for fixed length strings may be an array type whose element type is an Ada character type. If that Ada character type is defined so as to use the number of bits per character used by the SQL encoding, then the restriction on P is precisely P'LENGTH = *L*. For variable length strings using fixed length encodings, if the definition of CHAR in the appropriate VARYING package is based on the type `Ada.Strings.Unbounded.Unbounded_String`, there is no restriction on *P*. Otherwise, a precise statement of the restriction on *P* is implementation-defined.

n) If the *i*-th <host parameter declaration> specifies a <data type> that is NUMERIC(*P*,*S*) for some <precision> *P* and <scale> *S*, then the Ada library unit package generated for the encompassing module shall contain a declaration equivalent to:

```
subtype Numeric_p_s is
  Interfaces.SQL.Numerics.Scale_s digits p;
```

The subtype mark in the *i*-th parameter specification shall specify this subtype.

o) If the *i*-th <host parameter declaration> specifies a <data type> that is SMALLINT, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.SMALLINT`.

p) If the *i*-th <host parameter declaration> specifies a <data type> that is INTEGER, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.INT`.

q) If the *i*-th <host parameter declaration> specifies a <data type> that is BIGINT, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.BIGINT`.

r) If the *i*-th <host parameter declaration> specifies a <data type> that is REAL, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.REAL`.

s) If the *i*-th <host parameter declaration> specifies a <data type> that is DOUBLE_PRECISION, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.DOUBLE_PRECISION`.

t) For every parameter,

Case:

i) If the parameter is an input parameter but not an output parameter, then the Ada parameter mode is **in**.

ii) If the parameter is an output parameter but not an input parameter, then the Ada parameter mode is **out**.

iii) If the parameter is both an input parameter and an output parameter, then the Ada parameter mode is **in out**.

iv) Otherwise, the Ada parameter mode is **in**, **out**, or **in out**.

u) The following Ada library unit renaming declaration exists:

```
with Interfaces.SQL;
package SQL_Standard renames Interfaces.SQL.
```

3) If the caller language of the <externally-invoked procedure> is C, then:

a) The declared type of an SQLSTATE host parameter shall be C char with length 6.

b) For each *i*, 1 (one) < $i \le n$, PDT_i shall not identify a data type listed in the “SQL data type” column of Table 17, “Data type correspondences for C”, for which the corresponding row in the “C data type” column is 'None'.

c) For each *i*, 1 (one) < $i \le n$, the type of the *i*-th host parameter shall be the data type listed in the “C data type” column of Table 17, “Data type correspondences for C”, for which the corresponding row in the “SQL data type” column is PDT_i.

4) If the caller language of the <externally-invoked procedure> is COBOL, then:

a) The declared type of an SQLSTATE host parameter shall be COBOL PICTURE X(5).

b) For each *i*, 1 (one) ≤ $i \le n$, PDT_i shall not identify a data type listed in the “SQL data type” column of Table 18, “Data type correspondences for COBOL”, for which the corresponding row in the “COBOL data type” column is 'None'.

c) For each i, 1 (one) < $i \le n$, the type of the i-th host parameter shall be the data type listed in the "COBOL data type" column of Table 18, "Data type correspondences for COBOL", for which the corresponding row in the "SQL data type" column is PDT_i.

5) If the caller language of the <externally-invoked procedure> is FORTRAN, then:

a) The declared type of an SQLSTATE host parameter shall be Fortran CHARACTER with length 5.

b) For each i, 1 (one) $\le i \le n$, PDT_i shall not identify a data type listed in the "SQL data type" column of Table 19, "Data type correspondences for Fortran", for which the corresponding row in the "Fortran data type" column is 'None'.

c) For each i, 1 (one) < $i \le n$, the type of the i-th host parameter shall be the data type listed in the "Fortran data type" column of Table 19, "Data type correspondences for Fortran", for which the corresponding row in the "SQL data type" column is PDT_i.

6) If the caller language of the <externally-invoked procedure> is M, then:

a) The declared type of an SQLSTATE host parameter shall be M character with maximum length greater than or equal to 5.

b) For each i, 1 (one) $\le i \le n$, PDT_i shall not identify a data type listed in the "SQL data type" column of Table 20, "Data type correspondences for M", for which the corresponding row in the "MUMPS data type" column is 'None'.

c) For each i, 1 (one) < $i \le n$, the type of the i-th host parameter shall be the data type listed in the "MUMPS data type" column of Table 20, "Data type correspondences for M", for which the corresponding row in the "SQL data type" column is PDT_i.

7) If the caller language of the <externally-invoked procedure> is PASCAL, then:

a) The declared type of an SQLSTATE host parameter shall be Pascal PACKED ARRAY[1..5] OF CHAR.

b) For each i, 1 (one) $\le i \le n$, PDT_i shall not identify a data type listed in the "SQL data type" column of Table 21, "Data type correspondences for Pascal", for which the corresponding row in the "Pascal data type" column is 'None'.

c) For each i, 1 (one) < $i \le n$, the type of the i-th host parameter shall be the data type listed in the "Pascal data type" column of Table 21, "Data type correspondences for Pascal", for which the corresponding row in the "SQL data type" column is PDT_i.

8) If the caller language of the <externally-invoked procedure> is PLI, then:

a) The declared type of an SQLSTATE host parameter shall be PL/I CHARACTER(5).

b) For each i, 1 (one) $\le i \le n$, PDT_i shall not identify a data type listed in the "SQL data type" column of Table 22, "Data type correspondences for PL/I", for which the corresponding row in the "PL/I data type" column is 'None'.

c) For each i, 1 (one) < $i \le n$, the type of the i-th host parameter shall be the data type listed in the "PL/I data type" column of Table 22, "Data type correspondences for PL/I", for which the corresponding row in the "SQL data type" column is PDT_i.

Access Rules

None.

General Rules

1) Let *EP*, *PD*, *PN*, *DT*, and *PI* be a *PROC*, a *DECL*, a *NAME*, a *TYPE*, and an *ARG* specified in an application of the General Rules of this Subclause. Let *P* be the host parameter corresponding to *PD*.

2) If the General Rules of this Subclause are being applied for the evaluation of input parameters, and *P* is either an input host parameter or both an input host parameter and an output host parameter, then

   Case:

   a) If *DT* identifies a CHARACTER(*L*) or CHARACTER VARYING(*L*) data type and the caller language of *EP* is C, then a reference to *PN* is implicitly treated as a character string type value in the specified character set in which the octets of *PI* are the corresponding octets of that value.

      When such a reference is evaluated,

      Case:

      i) If *DT* identifies a CHARACTER(*L*) data type and some C character preceding the least significant C character of the value *PI* contains the implementation-defined null character that terminates a C character string, then the remaining characters of the value are set to <space>s.

      ii) If *DT* identifies a CHARACTER VARYING(*L*), then the length in characters of the value is set to the number of characters of PI_i that precede the implementation-defined null character that terminates a C character string.

      iii) If the least significant C character of the value *PI* does not contain the implementation-defined null character that terminates a C character string, then an exception condition is raised: *data exception — unterminated C string*; otherwise, that least significant C character does not correspond to any character in PI_i and is ignored.

   b) If *DT* identifies a CHARACTER(*L*) data type and the caller language of *EP* is either COBOL, FORTRAN, or PASCAL, or *DT* identifies a CHARACTER VARYING(*L*) data type and the caller language of *EP* is M, or *DT* identifies a CHARACTER(*L*) data type or CHARACTER VARYING(*L*) data type and the caller language of *EP* is PLI, then a reference to *PN* is implicitly treated as a character string type value in the specified character set in which the octets of *PI* are the corresponding octets of that value.

      NOTE 365 — In the preceding 2 Rules, the phrase "implementation-defined null character that terminates a C character string" implies one or more octets all of whose bits are zero and whose number is equal to the number of octets in the largest character of the character set of *DT*.

   c) If *DT* identifies INT, DEC, or REAL and the caller language of *EP* is M, then a reference to *PN* is implicitly treated as:

      ```
      CAST ( PI AS DT )
      ```

   d) If *DT* identifies a BOOLEAN type, then

      Case:

i) If the caller language of *EP* is ADA, then if *PI* is False, then a reference to *PN* has the value *False*; otherwise, a reference to *PN* has the value *True*.

ii) If the caller language of *EP* is C, then if *PI* is 0 (zero), then a reference to *PN* has the value *False*; otherwise, a reference to *PN* has the value *True*.

iii) If the caller language of *EP* is COBOL, then if *PI* is 'F', then a reference to *PN* has the value *False*; otherwise, a reference to *PN* has the value *True*.

iv) If the caller language of *EP* is FORTRAN, then if *PI* is .FALSE., then a reference to *PN* has the value *False*; otherwise, a reference to *PN* has the value *True*.

v) If the caller language of *EP* is PLI, then if *PI* is '0'B, then a reference to *PN* has the value *False*; otherwise, a reference to *PN* has the value *True*.

NOTE 366 — Pascal has a Boolean-type whose values are *True* and *False*.

e) If *P* is a binary large object locator parameter, a character large object locator parameter, an array locator parameter, a multiset locator parameter, or a user-defined type locator parameter, then a reference to *PN* in a <general value specification> has the corresponding large object value, the large object character string value, the array value, the multiset value, or the user-defined type value, respectively, corresponding to *PI*.

f) If *DT* identifies a CHARACTER LARGE OBJECT or BINARY LARGE OBJECT type, then

Case:

i) If the caller language of *EP* is C, then a reference to *PN* is implicitly treated as

Case:

1) If *DT* identifies a CHARACTER LARGE OBJECT type, then a character string containing the *PN*.*PN*_length characters of *PN*.*PN*_data starting at character number 1 (one) in the same order that the characters appear in *PN*.*PN*_data.

2) If *DT* identifies a BINARY LARGE OBJECT type, then a binary large object string containing the *PN*.*PN*_length octets of *PN*.*PN*_data starting at octet number 1 (one) in the same order that the octets appear in *PN*.*PN*_data.

ii) If the caller language of *EP* is COBOL, then a reference to *PN* is implicitly treated as

Case:

1) If *DT* identifies a CHARACTER LARGE OBJECT type, then a character string containing the *PN*.*PN*_length characters of *PN*.*PN*_data starting at character number 1 (one) in the same order that the characters appear in *PN*.*PN*_data.

2) If *DT* identifies a BINARY LARGE OBJECT type, then a binary large object string containing the *PN*.*PN*_length octets of *PN*.*PN*_data starting at octet number 1 (one) in the same order that the octets appear in *PN*.*PN*_data.

iii) If the caller language of *EP* is FORTRAN, then a reference to *PN* is implicitly treated as

Case:

1) If *DT* identifies a CHARACTER LARGE OBJECT type, then a character string containing the *PN*.*PN*_length characters of *PN*.*PN*_data starting at character number 1 (one) in the same order that the characters appear in *PN*.*PN*_data.

2) If *DT* identifies a BINARY LARGE OBJECT type, then a binary large object string containing the *PN*.*PN*_length octets of *PN*.*PN*_data starting at octet number 1 (one) in the same order that the octets appear in *PN*.*PN*_data.

iv) If the caller language of *EP* is PLI, then a reference to *PN* is implicitly treated as

Case:

1) If *DT* identifies a CHARACTER LARGE OBJECT type, then a character string containing the *PN*.*PN*_length characters of *PN*.*PN*_data starting at character number 1 (one) in the same order that the characters appear in *PN*.*PN*_data.

2) If *DT* identifies a BINARY LARGE OBJECT type, then a binary large object string containing the *PN*.*PN*_length octets of *PN*.*PN*_data starting at octet number 1 (one) in the same order that the octets appear in *PN*.*PN*_data.

g) If *DT* identifies a BINARY(*L*) or BINARY VARYING(*L*) data type and the caller language of *EP* is C, then a reference to *PN* is implicitly treated as a binary string type value in which the octets of *PI* are the corresponding octets of that value.

When such a reference is evaluated,

Case:

i) If *DT* identifies a BINARY(*L*) data type and the length in octets *LPI* of *PI* is less than *L*, then a reference to the remaining *L*–*LPI* octets of *PN* has the value X'00'.

ii) If *DT* identifies a BINARY VARYING(*L*), then a reference to *PN* is treated as a binary string containing the *PN*.*PN*_length octets of *PN*.*PN*_data starting at octet number 1 (one) in the same order that the octets appear in *PN*.*PN*_data.

h) If *DT* identifies a BINARY(*L*) data type and the caller language of *EP* is either COBOL or FORTRAN, or *DT* identifies a BINARY(*L*) data type or BINARY VARYING(*L*) data type and the caller language of *EP* is PLI, then a reference to *PN* is implicitly treated as a binary string type value in which the octets of *PI* are the corresponding octets of that value.

i) Otherwise, a reference to *PN* in a <general value specification> has the value *PI*.

3) If the General Rules of this Subclause are being applied for the evaluation of output parameters, and *P* is either an output host parameter or both an input host parameter and an output host parameter, then

Case:

a) If *DT* identifies CHARACTER(*L*) or CHARACTER VARYING(*L*) data types and the caller language of *EP* is C, then let *CL* be *k* greater than the maximum possible length in octets of *PN*, where *k* is the size in octets of the largest character in the character set of *DT*. A reference to *PN* that assigns some value *SV* to *PN* implicitly assigns a value that is an SQL CHARACTER(*CL*) data type in which octets of the value are the corresponding octets of SV_i, padded on the right with <space>s as necessary to reach the length *CL*, concatenated with a single implementation-defined null character that terminates a C character string.

b) If *DT* identifies a CHARACTER(*L*) data type and the caller language of *EP* is either COBOL, FORTRAN, or PASCAL, then let *CL* be the maximum possible length in octets of *PN*. A reference to *PN* that assigns some value *SV* to *PN* implicitly assigns a value that is an SQL CHARACTER(*CL*) data type in which octets of the value are the corresponding octets of *SV*, padded on the right with <space>s as necessary to reach the length *CL*.

c) If *DT* identifies a CHARACTER VARYING(*L*) data type and the caller language of *EP* is M, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns a value that is an SQL CHARACTER VARYING(*ML*) data type in which octets of the value are the corresponding octets of *SV*, padded on the right with <space>s as necessary to reach the length *CL*. *ML* is the implementation-defined maximum length of variable-length character strings.

d) If *DT* identifies a CHARACTER(*L*) or CHARACTER VARYING(*L*) data types and the caller language of *EP* is PLI, then let *CL* be the maximum possible length in octets of *PN*. A reference to *PN* that assigns some value *SV* to *PN* implicitly assigns a value that is

Case:

i) If *DT* identifies CHARACTER(*L*), then an SQL CHARACTER(*CL*) data type.

ii) Otherwise, an SQL CHARACTER VARYING(*CL*) data type in which octets of the value are the corresponding octets of *SV*, padded on the right with <space>s as necessary to reach the length *CL*.

NOTE 367 — In the preceding 4 Rules, the phrase "implementation-defined null character that terminates a C character string" implies one or more octets all of whose bits are zero and whose number is equal to the number of octets in the largest character of the character set of *DT*.

e) If *DT* identifies INT, DEC, or REAL and the caller language of *EP* is M, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value

```
CAST ( SV AS CHARACTER VARYING(ML) )
```

to *PI*, where *ML* is the implementation-defined maximum length of variable-length character strings.

f) If *DT* identifies a BOOLEAN type, then

Case:

i) If the caller language of *EP* is ADA, then a reference to *PN* that assigns the value *False* to *PN* implicitly assigns the False to *PI*; a reference to *PN* that assigns the value *True* implicitly assigns the value True to *PI*.

ii) If the caller language of *EP* is C, then a reference to *PN* that assigns the value *False* to *PN* implicitly assigns the value 0 (zero) to *PI*; a reference to *PN* that assigns the value *True* implicitly assigns the value 1 (one) to *PI*.

iii) If the caller language of *EP* is COBOL, then a reference to *PN* that assigns the value *False* to *PN* implicitly assigns the value 'F' to *PI*; a reference to *PN* that assigns the value *True* implicitly assigns the value 'T' to *PI*.

iv) If the caller language of *EP* is FORTRAN, then a reference to *PN* that assigns the value *False* to *PN* implicitly assigns the value .FALSE. to *PI*; a reference to *PN* that assigns the value *True* implicitly assigns the value .TRUE. to *PI*.

v) If the caller language of *EP* is PLI, then a reference to *PN* that assigns the value *False* to *PN* implicitly assigns the value '0'B to *PI*; a reference to *PN* that assigns the value *True* implicitly assigns the value '1'B to *PI*.

NOTE 368 — Pascal has a Boolean-type, whose values are *True* and *False*.

g) If *P* is a binary large object locator parameter, a character large object locator parameter, an array locator parameter, a multiset locator parameter, or a user-defined type locator parameter, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the corresponding large object locator value,

the character large object locator value, the array locator value, the multiset locator value, or the user-defined type locator value, respectively, that uniquely identifies *SV* to *PI*.

h) If *DT* identifies a CHARACTER LARGE OBJECT or BINARY LARGE OBJECT type, then

Case:

i) If the caller language of *EP* is C, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value LENGTH(*SV*) to *PN*.*PN*_length and the value *SV* to *PN*.*PN*_data.

ii) If the caller language of *EP* is COBOL, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value LENGTH(*SV*) to *PN*.*PN*-LENGTH and the value *SV* to *PN*.*PN*-DATA.

iii) If the caller language of *EP* is FORTRAN, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value LENGTH(*SV*) to *PN*_LENGTH and the value *SV* to *PN*_DATA.

iv) If the caller language of *EP* is PLI, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value LENGTH(*SV*) to *PN*.*PN*_length and the value *SV* to *PN*.*PN*_data.

i) If *DT* identifies a BINARY(*L*) data type and the caller language of *EP* is C, then let *CL* be the maximum possible length in octets of *PN*. A reference to *PN* that assigns some value *SV* to *PN* implicitly assigns a value that is an SQL BINARY(*CL*) data type in which octets of the value are the corresponding octets of SV_i, padded on the right with X'00's as necessary to reach the length *CL*.

j) If *DT* identifies a BINARY VARYING(*L*) data type and the caller language of *EP* is C, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value LENGTH(*SV*) to *PN*.*PN*_length and the value *SV* to *PN*.*PN*_data.

k) If *DT* identifies a BINARY(*L*) data type and the caller language of *EP* is either COBOL or FORTRAN, then let *CL* be the maximum possible length in octets of *PN*. A reference to *PN* that assigns some value *SV* to *PN* implicitly assigns a value that is an SQL BINARY(*CL*) data type in which octets of the value are the corresponding octets of *SV*, padded on the right with X'00's as necessary to reach the length *CL*.

l) If *DT* identifies a BINARY VARYING(*L*) data type and the caller language of *EP* is C, then a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value LENGTH(*SV*) to *PN*.*PN*-LENGTH and the value *SV* to *PN*.*PN*-DATA.

m) If *DT* identifies a BINARY(*L*) or BINARY VARYING(*L*) data types and the caller language of *EP* is PLI, then let *CL* be the maximum possible length in octets of *PN*. A reference to *PN* that assigns some value *SV* to *PN* implicitly assigns a value that is

Case:

i) If *DT* identifies BINARY(*L*), then an SQL BINARY(*CL*) data type.

ii) Otherwise, an SQL BINARY VARYING(*CL*) data type in which octets of the value are the corresponding octets of *SV*, padded on the right with X'00's as necessary to reach the length *CL*.

n) Otherwise, a reference to *PN* that assigns some value *SV* to *PN* implicitly assigns the value *SV* to *PI*. If the caller language of *EP* is ADA and no value has been assigned to *PI*, then an implementation-dependent value is assigned to *PI*.

Conformance Rules

None.

13.5 <SQL procedure statement>

This Subclause is modified by Subclause 11.2, "<SQL procedure statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 14.4, "<SQL procedure statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 13.2, "<SQL-procedure statement>", in ISO/IEC 9075-14.

Function

Define all of the SQL-statements that are <SQL procedure statement>s.

Format

```
<SQL procedure statement> ::=
  <SQL executable statement>

<SQL executable statement> ::=
    <SQL schema statement>
  | <SQL data statement>
  | <SQL control statement>
  |<SQL transaction statement>
  | <SQL connection statement>
  | <SQL session statement>
  | <SQL diagnostics statement>
  | <SQL dynamic statement>

<SQL schema statement> ::=
    <SQL schema definition statement>
  | <SQL schema manipulation statement>

<SQL schema definition statement> ::=
    <schema definition>
  | <table definition>
  | <view definition>
  | <SQL-invoked routine>
  | <grant statement>
  | <role definition>
  | <domain definition>
  | <character set definition>
  | <collation definition>
  | <transliteration definition>
  | <assertion definition>
  | <trigger definition>
  | <user-defined type definition>
  | <user-defined cast definition>
  | <user-defined ordering definition>
  | <transform definition>
  | <sequence generator definition>

<SQL schema manipulation statement> ::=
    <drop schema statement>
  | <alter table statement>
  | <drop table statement>
  | <drop view statement>
  | <alter routine statement>
```

```
  | <drop routine statement>
  | <drop user-defined cast statement>
  | <revoke statement>
  | <drop role statement>
  | <alter domain statement>
  | <drop domain statement>
  | <drop character set statement>
  | <drop collation statement>
  | <drop transliteration statement>
  | <drop assertion statement>
  | <drop trigger statement>
  | <alter type statement>
  | <drop data type statement>
  | <drop user-defined ordering statement>
  | <alter transform statement>
  | <drop transform statement>
  | <alter sequence generator statement>
  | <drop sequence generator statement>

<SQL data statement> ::=
    <open statement>
  | <fetch statement>
  | <close statement>
  | <select statement: single row>
  | <free locator statement>
  | <hold locator statement>
  | <SQL data change statement>

<SQL data change statement> ::=
    <delete statement: positioned>
  | <delete statement: searched>
  | <insert statement>
  | <update statement: positioned>
  | <update statement: searched>
  | <merge statement>

<SQL control statement> ::=
    <call statement>
  | <return statement>

<SQL transaction statement> ::=
    <start transaction statement>
  | <set transaction statement>
  | <set constraints mode statement>
  | <savepoint statement>
  | <release savepoint statement>
  | <commit statement>
  | <rollback statement>

<SQL connection statement> ::=
    <connect statement>
  | <set connection statement>
  | <disconnect statement>

<SQL session statement> ::=
    <set session user identifier statement>
  | <set role statement>
  | <set local time zone statement>
```

```
  | <set session characteristics statement>
  | <set catalog statement>
  | <set schema statement>
  | <set names statement>
  | <set path statement>
  | <set transform group statement>
  | <set session collation statement>

<SQL diagnostics statement> ::=
  <get diagnostics statement>

<SQL dynamic statement> ::=
    <SQL descriptor statement>
  | <prepare statement>
  | <deallocate prepared statement>
  | <describe statement>
  | <execute statement>
  | <execute immediate statement>
  | <SQL dynamic data statement>

<SQL dynamic data statement> ::=
    <allocate cursor statement>
  | <dynamic open statement>
  | <dynamic fetch statement>
  | <dynamic close statement>
  | <dynamic delete statement: positioned>
  | <dynamic update statement: positioned>

<SQL descriptor statement> ::=
    <allocate descriptor statement>
  | <deallocate descriptor statement>
  | <set descriptor statement>
  | <get descriptor statement>
```

Syntax Rules

1) Let *S* be the <SQL procedure statement>.

2) An <SQL connection statement> shall not be generally contained in an <SQL control statement>.

3) The SQL-invoked routine specified by <SQL-invoked routine> shall be a schema-level routine.

   NOTE 369 — "schema-level routine" is defined in Subclause 11.50, "<SQL-invoked routine>".

4) *S* is *possibly non-deterministic* if and only if *S* is not an <SQL schema statement> and at least one of the following is satisfied:

   a) *S* is a <select statement: single row> that is possibly non-deterministic.

   b) *S* contains a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.

   c) *S* generally contains a <query specification> or a <query expression> that is possibly non-deterministic.

   d) *S* generally contains a <value expression> that is possibly non-deterministic.

Access Rules

None.

General Rules

1) Let *S* be the executing statement specified in an application of this Subclause.

NOTE 370 — *S* is necessarily the innermost executing statement of the SQL-session as defined in Subclause 4.37, "SQL-sessions".

NOTE 371 — *S* is not necessarily an <SQL procedure statement>.

2) A *statement execution context NEWSEC* is established for the execution of *S*. Let *OLDSEC* be the most recent statement execution context. *NEWSEC* becomes the most recent statement execution context. *NEWSEC* is an atomic execution context, and therefore the most recent atomic execution context, if and only if *S* is an atomic SQL-statement.

3) The current trigger execution context *CTEC*, if any, is preserved and a new trigger execution context *NTEC* is created with an empty set of state changes *SSC*.

4) If the non-dynamic or dynamic execution of an <SQL data statement>, <SQL dynamic data statement>, <dynamic select statement>, or <dynamic single row select statement> occurs within the same SQL-transaction as the non-dynamic or dynamic 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*.

5) Case:

   a) If *S* is immediately contained in an <externally-invoked procedure> *EP*, then let *n* be the number of <host parameter declaration>s specified in *EP*; let PD_i, 1 (one) $\le i \le n$, be the *i*-th such <host parameter declaration>; and let PN_i and DT_i be the <host parameter name> and <data type>, respectively, specified in PD_i. When *EP* is called by an SQL-agent, let PI_i be the *i*-th argument in the procedure call.

      Case:

      i) If *S* is an <SQL connection statement>, then:

         1) The SQL-client module that contains *S* is associated with the SQL-agent.

         2) The first diagnostics area is emptied.

         3) For each *i*, 1 (one) $\le i \le n$, the General Rules of Subclause 13.4, "Calls to an <externally-invoked procedure>", are evaluated for input parameters with *EP*, PD_i, PN_i, DT_i, and PI_i as *PROC*, *DECL*, *NAME*, *TYPE*, and *ARG*, respectively.

         4) The General Rules of *S* are evaluated.

         5) For each *i*, 1 (one) $\le i \le n$, the General Rules of Subclause 13.4, "Calls to an <externally-invoked procedure>", are evaluated for output parameters with *EP*, PD_i, PN_i, DT_i, and PI_i as *PROC*, *DECL*, *NAME*, *TYPE*, and *ARG*, respectively.

         6) If *S* successfully initiated or resumed an SQL-session, then subsequent calls to an <externally-invoked procedure> and subsequent invocations of <direct SQL statement>s by the SQL-

agent are associated with that SQL-session until the SQL-agent terminates the SQL-session or makes it dormant.

ii) If *S* is an <SQL diagnostics statement>, then:

1) The SQL-client module that contains *S* is associated with the SQL-agent.

2) For each *i*, 1 (one) ≤ *i* ≤ *n*, the General Rules of Subclause 13.4, "Calls to an <externally-invoked procedure>", are evaluated for input parameters with *EP*, PD_i, PN_i, DT_i, and PI_i as *PROC*, *DECL*, *NAME*, *TYPE*, and *ARG*, respectively.

3) The General Rules of *S* are evaluated.

4) For each *i*, 1 (one) ≤ *i* ≤ *n*, the General Rules of Subclause 13.4, "Calls to an <externally-invoked procedure>", are evaluated for output parameters with *EP*, PD_i, PN_i, DT_i, and PI_i as *PROC*, *DECL*, *NAME*, *TYPE*, and *ARG*, respectively.

iii) Otherwise:

1) If no SQL-session is current for the SQL-agent, then

Case:

A) 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
```

B) 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
```

C) Otherwise, an exception condition is raised: *connection exception — connection does not exist*.

2) Subsequent calls to an <externally-invoked procedure> and subsequent invocations of <direct SQL statement>s by the SQL-agent are associated with the SQL-session until the SQL-agent terminates the SQL-session or makes it dormant.

3) If no SQL-transaction is active for the SQL-agent and *S* is an SQL-statement that implicitly initiates SQL-transactions, then an SQL-transaction is initiated. The SQL-client module that contains *S* is associated with the SQL-transaction.

4) The SQL-client module that contains *S* is associated with the SQL-agent.

5) 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*.

6) The first diagnostics area is emptied.

7) For each i, 1 (one) $\leq i \leq n$, the General Rules of Subclause 13.4, "Calls to an <externally-invoked procedure>", are evaluated for input parameters with *EP*, PD_i, PN_i, DT_i, and PI_i as *PROC*, *DECL*, *NAME*, *TYPE*, and *ARG*, respectively.

8) If *S* does not conform to the Syntax Rules and Access Rules of an <SQL procedure statement>, then an exception condition is raised: *syntax error or access rule violation*.

9) The General Rules of *S* are evaluated.

10) For each i, 1 (one) $\leq i \leq n$, the General Rules of Subclause 13.4, "Calls to an <externally-invoked procedure>", are evaluated for output parameters with *EP*, PD_i, PN_i, DT_i, and PI_i as *PROC*, *DECL*, *NAME*, *TYPE*, and *ARG*, respectively.

11) If *S* is a <select statement: single row> or a <fetch statement> and a completion condition is raised: *no data*, or an exception condition is raised, then the value of each PI_i for which PN_i is referenced in a <target specification> in *S* is implementation-dependent.

b) Otherwise:

i) If no SQL-transaction is active for the SQL-agent and *S* is an SQL-statement that implicitly initiates SQL-transactions, then an SQL-transaction is initiated.

ii) If *S* is 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*.

iii) If *S* is not an <SQL diagnostics statement>, then the first diagnostics area is emptied.

6) For every referential constraint whose constraint mode in the current SQL-session is immediate, the General Rules of Subclause 11.8, "<referential constraint definition>", are applied.

7) For every state change *DSC* in *SSC* whose trigger event is DELETE, let *DSOT* be the set of transitions in *DSC* and let *DBT* be the subject table of *DSC*. Each row that is marked for deletion from *DBT* is deleted from *DBT*.

8) Case:

a) If, for some constraint *C*, the constraint mode of *C* in the current SQL-session is immediate and an applicable <search condition> included in *CD* evaluates to *False*, then an exception condition is raised: *integrity constraint violation*.

NOTE 372 — This rule implies a re-evaluation of the applicable <search condition>s of referential constraints. This is needed to cater for both referential constraints specifying NO ACTION and referential constraints that are not satisfied after application of SET .

b) Otherwise, the Syntax Rules and General Rules of Subclause 15.17, "Execution of AFTER triggers", are applied with *SSC* as the *SET OF STATE CHANGES*.

9) *NTEC*, together with all of its contents, is destroyed and *CTEC*, if present, is restored to become the current trigger execution context.

10) Case:

a) If *S* is immediately contained in an <externally-invoked procedure>, then

Case:

i) 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*, as determined by the General Rules in this and other Subclauses of ISO/IEC 9075.

ii) If *S* did not execute successfully, then:

1) The status parameter is set to the value specified for the condition in Clause 24, "Status codes".

2) If *S* is an atomic SQL-statement, then all changes made to SQL-data or schemas by the execution of *S* are canceled.

NOTE 373 — Atomic and non-atomic SQL-statements are defined in Subclause 4.33.5, "SQL-statement atomicity and statement execution contexts".

b) Otherwise, the General Rules for *S* are evaluated.

Case:

i) 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*, as determined by the General Rules in this and other Subclauses of ISO/IEC 9075.

ii) Otherwise:

1) If *S* is an atomic SQL-statement, then all changes made to SQL-data or schemas by the execution of *S* are canceled.

NOTE 374 — Atomic and non-atomic SQL-statements are defined in Subclause 4.33.5, "SQL-statement atomicity and statement execution contexts".

2) The exception condition with which the execution of *S* completed is raised.

11) If *S* is not an <SQL diagnostics statement>, then 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. Whether any other condition areas become occupied is implementation-defined.

12) If *NEWSEC* is atomic, then all savepoints established during its existence are destroyed.

13) *NEWSEC* ceases to exist and *OLDSEC* becomes the most recent statement execution context.

14) *S* ceases to be an executing statement.

NOTE 375 — The innermost executing statement, if any, is now the one that was the innermost executing statement that caused *S* to be executed.

Conformance Rules

None.

13.6 Data type correspondences

This Subclause is modified by Subclause 14.5, "Data type correspondences", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 13.3, "Data type correspondences", in ISO/IEC 9075-14.

Function

Specify the data type correspondences for SQL data types and host language types.

NOTE 376 — These tables are referenced in Subclause 11.50, "<SQL-invoked routine>", for the definitions of external routines and in Subclause 10.4, "<routine invocation>", for the invocation of external routines.

In the following tables, let *P* be <precision>, *S* be <scale>, *L* be <length>, *T* be <time fractional seconds precision>, *Q* be <interval qualifier>, and *N* be the implementation-defined size of a structured type reference.

Tables

Table 16 — Data type correspondences for Ada

SQL Data Type	Ada Data Type
SQLSTATE	SQL_STANDARD.SQLSTATE_TYPE
CHARACTER (*L*)	SQL_STANDARD.CHAR, with P'LENGTH of *L*
CHARACTER VARYING (*L*)	*None*
CHARACTER LARGE OBJECT (*L*)	*None*
BINARY	*None*
BINARY VARYING	*None*
BINARY LARGE OBJECT(*L*)	*None*
NUMERIC(*P*,*S*)	*None*
DECIMAL(*P*,*S*)	*None*
SMALLINT	SQL_STANDARD.SMALLINT
INTEGER	SQL_STANDARD.INT
BIGINT	SQL_STANDARD.BIGINT
FLOAT(*P*)	*None*
REAL	SQL_STANDARD.REAL

SQL Data Type	Ada Data Type
DOUBLE PRECISION	SQL_STANDARD.DOUBLE_PRECISION
BOOLEAN	SQL_STANDARD.BOOLEAN
DATE	*None*
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
INTERVAL(*Q*)	*None*
user-defined type	*None*
REF	SQL_STANDARD.CHAR with P'LENGTH of *N*
ROW	*None*
ARRAY	*None*
MULTISET	*None*

Table 17 — Data type correspondences for C

SQL Data Type	C Data Type
SQLSTATE	`char`, with length 6
CHARACTER (*L*)[3]	`char`, with length (*L*+1)*k[1]
CHARACTER VARYING (*L*)[3]	`char`, with length (*L*+1)*k[1]

CHARACTER LARGE OBJECT (*L*)	`struct {`
	`  long hvn_reserved`
	`  unsigned long hvn_length`
	`  char hvn_data[L];`
	`} hvn`[2] [3]

BINARY (*L*)	`char`, with length *L*

BINARY VARYING (*L*)	`struct {`
	`  long hvn_reserved`
	`  unsigned long hvn_length`
	`  char hvn_data[L];`
	`} hvn`[2]

SQL Data Type	C Data Type

BINARY LARGE OBJECT (*L*)	`struct {`
	`  long hvn_reserved`
	`  unsigned long hvn_length`
	`  char hvn_data[L];`
	`} hvn`[2]

NUMERIC (*P*,*S*)	*None*
DECIMAL (*P*,*S*)	*None*
SMALLINT	pointer to `short`
INTEGER	pointer to `long`
BIGINT	pointer to `long long`
FLOAT (*P*)	*None*
REAL	pointer to `float`
DOUBLE PRECISION	pointer to `double`
BOOLEAN	pointer to `long`
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
user-defined type	*None*
REF	`char`, with length *N*
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] For character set UTF16, as well as other implementation-defined character sets in which a code unit occupies two octets, *k* is the length in units of C **unsigned short** of the character encoded using the greatest number of such units in the character set; for character set UTF32, as well as other implementation-defined character sets in which a code unit occupies four octets, *k* is four; for other character sets, *k* is the length in units of C **char** of the character encoded using the greatest number of such units in the character set.

[2] *hvn* is the name of the host variable defined to correspond to the SQL data type

[3] For character set UTF16, as well as other implementation-defined character sets in which a code unit occupies two octets, **char** or **unsigned char** should be replaced with **unsigned short**; for character set UTF32, as well as other implementation-defined

SQL Data Type	C Data Type
character sets in which a code unit occupies four octets, **char** or **unsigned char** should be replaced with **unsigned int**. Otherwise, **char** or **unsigned char** should be used.	

Table 18 — Data type correspondences for COBOL

SQL Data Type	COBOL Data Type
SQLSTATE	PICTURE X(5)
CHARACTER (*L*)	PICTURE X(*L*)[3]
CHARACTER VARYING (*L*)	*None*

CHARACTER LARGE OBJECT (*L*)	`01 hvn.`
	`  49 hvn-RESERVED PIC S9(9) USAGE IS BINARY.`
	`  49 hvn-LENGTH PIC S9(9) USAGE IS BINARY.`
	`  49 hvn-DATA PIC X(L)`[2] [3]`.`

BINARY (*L*)	PICTURE X(*L*)
BINARY VARYING (*L*)	*None*

BINARY LARGE OBJECT (*L*)	`01 hvn.`
	`  49 hvn-RESERVED PIC S9(9) USAGE IS BINARY.`
	`  49 hvn-LENGTH PIC S9(9) USAGE IS BINARY.`
	`  49 hvn-DATA PIC X(L)`[2]`.`

NUMERIC (*P*,*S*)	USAGE DISPLAY SIGN LEADING SEPARATE, with PICTURE as specified[1]
DECIMAL(*P*,*S*)	*None*
SMALLINT	PICTURE S9(*SPI*) USAGE BINARY, where *SPI* is implementation-defined
INTEGER	PICTURE S9(*PI*) USAGE BINARY, where *PI* is implementation-defined
BIGINT	PICTURE S9(*BPI*) USAGE BINARY, where *BPI* is implementation-defined
FLOAT (*P*)	*None*
REAL	*None*
DOUBLE PRECISION	*None*
BOOLEAN	PICTURE X

SQL Data Type	COBOL Data Type
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
user-defined type	*None*
REF	alphanumeric with length *N*
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] Case:

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

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

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

[2] *hvn* is the name of the host variable defined to correspond to the SQL data type

[3] For character set UTF16, as well as other implementation-defined character sets in which a code unit occupies two octets, "PICTURE X(*L*)" should be replaced with "PICTURE N(*L*)". For character set UTF32, as well as other implementation-defined character sets in which a code unit occupies four octets, "PICTURE X(*L*)" should be replaced with "PICTURE (*????*)". Otherwise, "PICTURE X(*L*)" should be used.

Table 19 — Data type correspondences for Fortran

SQL Data Type	Fortran Data Type
SQLSTATE	CHARACTER, with length 5
CHARACTER (*L*)	CHARACTER[2], with length *L*
CHARACTER VARYING (*L*)	*None*

CHARACTER LARGE OBJECT (*L*)	`CHARACTER hvn(L+8)`
	`INTEGER*4 hvn_RESERVED`
	`INTEGER*4 hvn_LENGTH`
	`CHARACTER hvn_DATA`
	`EQUIVALENCE(hvn(5), hvn_LENGTH)`
	`EQUIVALENCE(hvn(9), hvn_DATA)`[1] [2]

BINARY (*L*)	CHARACTER, with length (*L*)

SQL Data Type	Fortran Data Type
BINARY VARYING (*L*)	*None*

BINARY LARGE OBJECT (*L*)	`CHARACTER hvn(L+8)`
	`INTEGER*4 hvn_RESERVED`
	`INTEGER*4 hvn_LENGTH`
	`CHARACTER hvn_DATA`
	`EQUIVALENCE(hvn(5), hvn_LENGTH)`
	`EQUIVALENCE(hvn(9), hvn_DATA)`[1]

NUMERIC (*P*,*S*)	*None*
DECIMAL (*P*,*S*)	*None*
SMALLINT	*None*
INTEGER	INTEGER
BIGINT	*None*
FLOAT (*P*)	*None*
REAL	REAL
DOUBLE PRECISION	DOUBLE PRECISION
BOOLEAN	LOGICAL
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
user-defined type	*None*
REF	CHARACTER with length *N*
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] *hvn* is the name of the host variable defined to correspond to the SQL data type

[2] For character set UTF16, as well as other implementation-defined character sets in which a code unit occupies more than one octet, “CHARACTER KIND=*n*” should be used; in this case, the value of *n* that corresponds to a given character set is implementation-defined. Otherwise, “CHARACTER” (without “KIND=*n*”) should be used.

Table 20 — Data type correspondences for M

SQL Data Type	M Data Type
SQLSTATE	character, with maximum length at least 5
CHARACTER (*L*)	*None*
CHARACTER VARYING (*L*)	character with maximum length *L*
CHARACTER LARGE OBJECT (*L*)	*None*
BINARY (*L*)	*None*
BINARY VARYING (*L*)	*None*
BINARY LARGE OBJECT (*L*)	*None*
NUMERIC (*P*,*S*)	character
DECIMAL (*P*,*S*)	character
SMALLINT	*None*
INTEGER	character
BIGINT	*None*
FLOAT (*P*)	*None*
REAL	character
DOUBLE PRECISION	*None*
BOOLEAN	*None*
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
user-defined type	*None*
REF	character
ROW	*None*

SQL Data Type	M Data Type
ARRAY	*None*
MULTISET	*None*

Table 21 — Data type correspondences for Pascal

SQL Data Type	Pascal Data Type
SQLSTATE	PACKED ARRAY [1..5] OF CHAR
CHARACTER (1)	CHAR
CHARACTER (*L*), *L* > 1	PACKED ARRAY [1..*L*] OF CHAR
CHARACTER VARYING (*L*)	*None*
CHARACTER LARGE OBJECT (*L*)	*None*
BINARY (*L*)	*None*
BINARY VARYING (*L*)	*None*
BINARY LARGE OBJECT (*L*)	*None*
NUMERIC (*P*,*S*)	*None*
DECIMAL (*P*,*S*)	*None*
SMALLINT	*None*
INTEGER	INTEGER
BIGINT	*None*
FLOAT (*P*)	*None*
REAL	REAL
DOUBLE PRECISION	*None*
BOOLEAN	BOOLEAN
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*

SQL Data Type	Pascal Data Type
INTERVAL (*Q*)	*None*
user-defined type	*None*
REF	PACKED ARRAY[1..*N*] OF CHAR
ROW	*None*
ARRAY	*None*
MULTISET	*None*

Table 22 — Data type correspondences for PL/I

SQL Data Type	PL/I Data Type
SQLSTATE	CHARACTER(5)
CHARACTER (*L*)	CHARACTER(*L*)
CHARACTER VARYING (*L*)	CHARACTER (*L*) VARYING

CHARACTER LARGE OBJECT (*L*)	`DCL 01 hvn`
	`  49 hvn_reserved FIXED BINARY (31)`
	`  49 hvn_length FIXED BINARY (31)`
	`  49 hvn_data CHAR (n)`[1]`;`

BINARY (*L*)	CHARACTER (*L*)
BINARY VARYING (*L*)	CHARACTER (*L*) VARYING

BINARY LARGE OBJECT (*L*)	`DCL 01 hvn`
	`  49 hvn_reserved FIXED BINARY (31)`
	`  49 hvn_length FIXED BINARY (31)`
	`  49 hvn_data CHAR (n)`[1]`;`

NUMERIC(*P*,*S*)	*None*
DECIMAL (*P*,*S*)	FIXED DECIMAL (*P*,*S*)
SMALLINT	FIXED BINARY(*SPI*), where *SPI* is implementation-defined
INTEGER	FIXED BINARY(*PI*), where *PI* is implementation-defined
BIGINT	FIXED BINARY(*BPI*), where *BPI* is implementation-defined
FLOAT (*P*)	FLOAT BINARY (*P*)

SQL Data Type	PL/I Data Type
REAL	*None*
DOUBLE PRECISION	*None*
BOOLEAN	BIT(1)
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
user-defined type	*None*
REF	CHARACTER VARYING(*N*)
ROW	*None*
ARRAY	*None*
MULTISET	*None*
[1] *hvn* is the name of the host variable defined to correspond to the SQL data type	

Conformance Rules

None.

(Blank page)

14 Data manipulation

This Clause is modified by Clause 12, "Data manipulation", in ISO/IEC 9075-4.
This Clause is modified by Clause 11, "<SQLJ specific clause> and contents", in ISO/IEC 9075-10.
This Clause is modified by Clause 14, "Data manipulation", in ISO/IEC 9075-14.

14.1 <declare cursor>

This Subclause is modified by Subclause 12.1, "<declare cursor>", in ISO/IEC 9075-4.

Function

Declare a standing cursor.

Format

```
<declare cursor> ::=
  DECLARE <cursor name> <cursor properties>
      FOR <cursor specification>
```

Syntax Rules

1) If a <declare cursor> is contained in an <SQL-client module definition> *M*, then:

   a) The <cursor name> shall not be equivalent to the <cursor name> of any other <declare cursor> or <dynamic declare cursor> in *M*.

   b) The scope of the <cursor name> is *M* with the exception of any <SQL schema statement> contained in *M*.

   c) Any <host parameter name> contained in the <cursor specification> shall be defined in a <host parameter declaration> in the <externally-invoked procedure> that contains an <open statement> that specifies the <cursor name> and is contained in the scope of that <cursor name>.

   NOTE 377 — See the Syntax Rules of Subclause 13.1, "<SQL-client module definition>".

Access Rules

None.

General Rules

1) A cursor declaration descriptor *CDD* is created. *CDD* includes indications that:

   a) The kind of cursor is a standing cursor.

   b) The provenance of the cursor is an indication of the SQL-client module whose <SQL-client module definition> contains the <declare cursor>.

   c) The name of the cursor is the <cursor name>.

   d) The cursor's origin is the <cursor specification> contained in the <declare cursor>.

   e) The cursor's declared properties are as determined by the <cursor properties>.

Conformance Rules

None.

14.2 <cursor properties>

Function

Specify the declared properties of a cursor.

Format

```
<cursor properties> ::=
  [ <cursor sensitivity> ] [ <cursor scrollability> ] CURSOR
      [ <cursor holdability> ]
      [ <cursor returnability> ]

<cursor sensitivity> ::=
    SENSITIVE
  | INSENSITIVE
  | ASENSITIVE

<cursor scrollability> ::=
    SCROLL
  | NO SCROLL

<cursor holdability> ::=
    WITH HOLD
  | WITHOUT HOLD

<cursor returnability> ::=
    WITH RETURN
  | WITHOUT RETURN
```

Syntax Rules

1) If <cursor sensitivity> is not specified, then ASENSITIVE is implicit.

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

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

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

Access Rules

None.

General Rules

1) The declared properties of the cursor declaration descriptor associated with <cursor properties> are given by:

   a) The declared sensitivity property is the explicit or implicit <cursor sensitivity>.

   b) The declared scrollability property is the explicit or implicit <cursor scrollability>.

c) The declared holdability property is the explicit or implicit <cursor holdability>.

d) The declared returnability property is the explicit or implicit <cursor returnability>.

Conformance Rules

1) Without Feature T231, “Sensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains SENSITIVE.

2) Without Feature F791, “Insensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains INSENSITIVE.

3) Without Feature F791, “Insensitive cursors”, or Feature T231, “Sensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains ASENSITIVE.

4) Without Feature F431, “Read-only scrollable cursors”, conforming SQL language shall not contain a <cursor scrollability>.

5) Without Feature T471, “Result sets return value”, conforming SQL language shall not contain a <cursor returnability>.

6) Without Feature T551, “Optional key words for default syntax”, conforming SQL language shall not contain a <cursor holdability> that immediately contains WITHOUT HOLD.

14.3 <cursor specification>

Function

Define a result set.

Format

```
<cursor specification> ::=
  <query expression> [ <order by clause> ] [ <updatability clause> ]

<updatability clause> ::=
  FOR { READ ONLY | UPDATE [ OF <column name list> ] }

<order by clause> ::=
  ORDER BY <sort specification list>
```

Syntax Rules

1) Let *CS* be the <cursor specification>.

2) Let *QE* be the <query expression> immediately contained in *CS*.

3) Case:

   a) If *CS* is contained in a <declare cursor> *DC* that contains a <cursor properties>, then let *CP* be that <cursor properties>.

   b) If *CS* is a <preparable statement> being prepared by a <prepare statement> or re-prepared by an <allocate cursor statement> or a <dynamic open statement>, then:

      i) Let *STMT* be the <prepare statement> that is preparing *CS*, or the <allocate cursor statement> or <dynamic open statement> that is re-preparing *CS*. Let *SCMD* be the <SQL-client module definition> that contains *STMT*.

      ii) Case:

         1) If *CS* is being prepared by a <prepare statement>, then let *PS* be that <prepare statement>.

         2) Otherwise, let *PS* be the <prepare statement> that previously prepared *CS*.

      iii) Let *SSV* be the <SQL statement variable> contained in *PS*.

      iv) Case:

         1) If *CS* is being prepared by a <prepare statement>, then:

            Case:

            A) If *SSV* is a <statement name> and there is exactly one <dynamic declare cursor> *DDC* contained in *SCMD* whose <statement name> is equivalent to *SSV*, then let *CP1* be the <cursor properties> contained in *DDC*.

B) If *SSV* is an <extended statement name> that specifies or implies a <scope option> that is LOCAL, and there is exactly one <allocate cursor statement> *ACS* contained in *SCMD* whose <cursor intent> is <statement cursor> and whose <extended statement name> specifies or implies LOCAL, then let *CP1* be the <cursor properties> contained in *ACS*.

C) Otherwise, let *CP1* be a zero-length string.

2) If *CS* is being re-prepared by an <allocate cursor statement> *ACS*, then let *CP1* be the <cursor properties> contained in *ACS*.

3) If CS is being re-prepared by a <dynamic open statement>, then let *DDC* be the <dynamic declare cursor> whose <cursor name> is equivalent to the <cursor name> contained in *STMT*. Let *CP1* be the <cursor properties> contained in *DDC*.

v) If *PS* contains an <attributes variable>, then let *CP2* be the value of that <attributes variable>; otherwise, let *CP2* be the zero-length string.

vi) Case:

1) If *CP2* contains <cursor sensitivity>, then let *SENS* be that <cursor sensitivity>.

2) If *CP1* contains <cursor sensitivity>, then let *SENS* be that <cursor sensitivity>.

3) Otherwise, let *SENS* be the zero-length string.

vii) Case:

1) If *CP2* contains <cursor scrollability>, then let *SCRO* be that <cursor scrollability>.

2) If *CP1* contains <cursor scrollability>, then let *SCRO* be that <cursor scrollability>.

3) Otherwise, let *SCRO* be the zero-length string.

viii) Case:

1) If *CP2* contains <cursor holdability>, then let *HOLD* be that <cursor holdability>.

2) If *CP1* contains <cursor holdability>, then let *HOLD* be that <cursor holdability>.

3) Otherwise, let *HOLD* be the zero-length string.

ix) Case:

1) If *CP2* contains <cursor returnability>, then let *RET* be that <cursor returnability>.

2) If *CP1* contains <cursor returnability>, then let *RET* be that <cursor returnability>.

3) Otherwise, let *RET* be the zero-length string.

x) Let *CP* be the <cursor properties>:

```
SENS SCRO CURSOR HOLD RET
```

4) If <updatability clause> is not specified and either *CS* is contained in a <declare cursor> or is being re-prepared by an <allocate cursor statement> or a <dynamic open statement>, then

Case:

a) If *CP* contains INSENSITIVE or SCROLL, or *CS* contains ORDER BY, or if *QE* is not a simply updatable <query specification>, then an <updatability clause> of READ ONLY is implicit.

b) Otherwise, an <updatability clause> of FOR UPDATE without a <column name list> is implicit.

NOTE 378 — If *CS* is being prepared by a <prepare statement>, then defaulting the <updatability clause> is postponed until *CS* is re-prepared.

5) If an <updatability clause> of FOR UPDATE with or without a <column name list> is specified, then *CP* shall not contain INSENSITIVE, *QE* shall be updatable, and *QE* shall have only one leaf underlying table *LUT* such that *QE* is one-to-one with respect to *LUT*.

6) Case:

a) If an <updatability clause> specifying FOR UPDATE is specified or implicit, then *CS* is *updatable*.

b) If an <updatability clause> specifying FOR READ ONLY is specified or implicit, then *CS* is *not updatable*.

c) Otherwise, the determination of updatability of *CS* is postponed until *CS* is re-prepared.

7) If *CS* is updatable, then let *LUTN* be a <table name> that references *LUT*. *LUTN* is an exposed <table or query name> whose scope is <updatability clause>.

8) If an <order by clause> is specified, then:

a) Let *OBC* be the <order by clause>. Let *NSK* be the number of <sort specification>s in *OBC*. For each *i* between 1 (one) and *NSK*, let K_i be the <sort key> contained in the *i*-th <sort specification> in *OBC*.

b) Each K_i shall contain a <column reference> and shall not contain a <query expression> or a <set function specification>.

c) If *QE* is a <query expression body> that is a <query term> that is a <query primary> that is a <simple table> that is a <query specification>, then the <cursor specification> is said to be a *simple table query*.

d) Case:

i) If <sort specification list> contains any <sort key> K_i that contains a column reference to a column that is not a column of *T*, then:

1) The <cursor specification> shall be a simple table query.

2) Let *TE* be the <table expression> immediately contained in the <query specification> *QS* contained in *QE*.

3) Let *SL* be the <select list> of *QS*. Let *SLT* be obtained from *SL* by replacing each <column reference> with its fully qualified equivalent.

4) Let *OBCT* be obtained from *OBC* by replacing each <column reference> that references a column of *TE* with its fully qualified equivalent; in the case of common column names, each common column name is regarded as fully qualified.

5) For each *i* between 1 (one) and *NSK*, let KT_i be the <sort key> contained in the *i*-th <sort specification> contained in *OBCT*.

6) For each *i* between 1 (one) and *NSK*, if KT_i has the same left normal form derivation as the <value expression> immediately contained in some <derived column> *DC* of *SLT*, then:

NOTE 379 — "Left normal form derivation" is defined in Subclause 6.2, "Notation provided in this International Standard", in ISO/IEC 9075-1.

A) Case:

I) If *DC* simply contains an <as clause>, then let *CN* be the <column name> contained in the <as clause>.

II) Otherwise, let *CN* be an implementation-dependent <column name> that is not equivalent to the explicit or implicit <column name> of any other <derived column> contained in *SLT*. Let *VE* be the <value expression> simply contained in *DC*. *DC* is replaced in *SLT* by

```
VE AS CN
```

B) KT_i is replaced in *OBCT* by

```
CN
```

7) Let *SCR* be the set of <column reference>s to columns of *TE* that remain in *OBCT* after the preceding transformation.

8) Let *NSCR* be the number of <column reference>s contained in *SCR*. For each *j* between 1 (one) and *NCR*, let C_j be an enumeration of these <column reference>s.

9) Case:

A) If *NSCR* is 0 (zero), then let *SKL* be the zero-length string.

B) Otherwise:

I) *T* shall not be a grouped table.

II) *QS* shall not specify the <set quantifier> DISTINCT or directly contain one or more <set function specification>s.

III) Let *SKL* be the comma-separated list of <derived column>s:

$$, C_1, C_2, \ldots, C_{NCR}$$

The columns C_j are said to be *extended sort key columns*.

10) Let *ST* be the result of evaluating the <query specification>:

```
SELECT SLT SKL TE
```

11) Let *EOBC* be *OBCT*.

ii) Otherwise, let *ST* be *T* and let *EOBC* be *OBC*.

e) *ST* is said to be the *sort table* of *CS*.

9) If an <updatability clause> of FOR UPDATE without a <column name list> is specified or implicit, then a <column name list> that consists of the <column name> of every column of *LUT* is implicit.

10) If an <updatability clause> of FOR UPDATE with a <column name list> is specified, then each <column name> in the <column name list> shall be the <column name> of a column of *LUT*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F831, “Full cursor update”, conforming SQL language shall not contain an <updatability clause> that contains FOR UPDATE and that contains a <cursor scrollability>.

2) Without Feature F831, “Full cursor update”, conforming SQL language shall not contain an <updatability clause> that specifies FOR UPDATE and that contains an <order by clause>.

3) Without Feature T111, “Updatable joins, unions, and columns”, in conforming SQL language, if FOR UPDATE is specified, then *QE* shall be simply updatable.

14.4 <open statement>

This Subclause is modified by Subclause 12.2, "<open statement>", in ISO/IEC 9075-4.

Function

Open a standing cursor.

Format

```
<open statement> ::=
  OPEN <cursor name>
```

Syntax Rules

1) Let *CN* be the <cursor name> in the <open statement>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified specified by *CN*.

Access Rules

1) The Access Rules for the <query expression> simply contained in the <declare cursor> identified by the <cursor name> are applied.

General Rules

1) Let *CR* be the cursor instance descriptor in the current SQL-session whose cursor declaration descriptor is *CDD*.

2) The General Rules of Subclause 15.1, "Effect of opening a cursor", are applied with *CR* as *CURSOR*.

Conformance Rules

None.

14.5 <fetch statement>

This Subclause is modified by Subclause 12.3, "<fetch statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.15, "<fetch statement>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 14.1, "<fetch statement>", in ISO/IEC 9075-14.

Function

Position a standing cursor on a specified row of the standing cursor's result set and retrieve values from that row.

Format

```
<fetch statement> ::=
  FETCH [ [ <fetch orientation> ] FROM ] <cursor name> INTO <fetch target list>

<fetch orientation> ::=
    NEXT
  | PRIOR
  | FIRST
  | LAST
  | { ABSOLUTE | RELATIVE } <simple value specification>

<fetch target list> ::=
  <target specification> [ { <comma> <target specification> }... ]
```

Syntax Rules

1) <fetch target list> shall not contain a <target specification> that specifies a <column reference>.

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

3) Let *CN* be the <cursor name> in the <fetch statement>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

4) *CN* shall identify a standing cursor.

5) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

6) Let *T* be the result set defined by the <cursor specification> of *CDD*.

7) If the implicit or explicit <fetch orientation> is not NEXT, then the declared scrollability property of *CDD* shall be SCROLL.

8) If a <fetch orientation> that contains a <simple value specification> is specified, then the declared type of that <simple value specification> shall be exact numeric with a scale of 0 (zero).

9) Case:

   a) If the <fetch target list> contains a single <target specification> *TS* and the degree of *T* is greater than 1 (one), then the declared type of *TS* shall be a row type.

      Case:

i) If *TS* is an <SQL parameter reference>, then the Syntax Rules of Subclause 9.2, "Store assignment", apply to *TS* as *TARGET* and the row type of *T* as *VALUE*.

ii) Otherwise, the Syntax Rules of Subclause 9.1, "Retrieval assignment", apply to *TS* as *TARGET* and the row type of *T* as *VALUE*.

b) Otherwise:

i) The number of <target specification>s *NTS* in the <fetch target list> shall be the same as the degree of *T*. The *i*-th <target specification>, 1 (one) ≤ *i*, ≤ *NTS*, in the <fetch target list> corresponds with the *i*-th column of *T*.

ii) For *i* varying from 1 (one) to *NTS*, let $TS1_i$ be the *i*-th <target specification> in the <fetch target list> that is either an <SQL parameter reference> or a <target array element specification>, and let CS_i be the *i*-th column of *T* that corresponds with the <target specification> in the <fetch target list>.

Case:

1) If $TS1_i$ contains a <simple value specification>, then the Syntax Rules of Subclause 9.2, "Store assignment", apply to an arbitrary site whose declared type is the declared type of $TS1_i$ and CS_i as *TARGET* and *VALUE*, respectively.

2) Otherwise, the Syntax Rules of Subclause 9.2, "Store assignment", apply to $TS1_i$ and the corresponding column of *T* as *TARGET* and *VALUE*, respectively.

iii) For each <target specification> $TS2_i$, 1 (one) ≤ *i*, ≤ *NTS*, that is a <host parameter specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", apply to $TS2_i$ as *TARGET* and the corresponding column of *T* as *VALUE*.

iv) For each <target specification> $TS2_i$, 1 (one) ≤ *i*, ≤ *NTS*, that is an <embedded variable specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", apply to $TS2_i$ as *TARGET* and the corresponding column of *T* as *VALUE*.

Access Rules

None.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) If *CR* is not in the open state, then an exception condition is raised: *invalid cursor state*.

3) The General Rules of Subclause 15.3, "Determination of the current row of a cursor", are applied with *CR* as *CURSOR* and <fetch orientation> as *FETCH ORIENTATION*.

4) If a completion condition *no data* has been raised, then no further General Rules of this Subclause are applied.

5) Case:

a) If the <fetch target list> contains a single <target specification> *TS* and the degree of *T* is greater than 1 (one), then the current row is assigned to *TS* and

Case:

i) If *TS* is an <SQL parameter reference>, then the General Rules of Subclause 9.2, "Store assignment", are applied to *TS* and the current row as *TARGET* and *VALUE*, respectively.

ii) Otherwise, the General Rules of Subclause 9.1, "Retrieval assignment", are applied to *TS* and the current row as *TARGET* and *VALUE*, respectively.

b) Otherwise, if the <fetch target list> contains more than one <target specification>, then values from the current row are assigned to their corresponding targets identified by the <fetch target list>. The assignments are made in an implementation-dependent order. Let *TV* be a target and let *SV* denote its corresponding value in the current row of *CR*.

Case:

i) If *TV* is either an <SQL parameter reference> or a <target array element specification>, then for each <target specification> in the <fetch target list>, let TV_i be the *i*-th <target specification> in the <fetch target list> and let SV_i denote the *i*-th corresponding value in the current row of *CR*.

Case:

1) If <target array element specification> is specified, then

Case:

A) If the value of TV_i, denoted by *C*, is the null value, then an exception condition is raised: *data exception — null value in array target*.

B) Otherwise:

I) Let *N* be the maximum cardinality of *C*.

II) Let *M* be the cardinality of the value of *C*.

III) Let *I* be the value of the <simple value specification> immediately contained in TV_i.

IV) Let *EDT* be the element type of *C*.

V) Case:

1) If *I* is greater than zero and less than or equal to *M*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *M* derived as follows:

a) For *j* varying from 1 (one) to *I*–1 and from *I*+1 to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

b) The *I*-th element of *A* is set to the value of *SV*, denoted by SV_i, by applying the General Rules of Subclause 9.2, "Store assignment", to the *I*-th element of *A* and SV_i as *TARGET* and *VALUE*, respectively.

2) If *I* is greater than *M* and less than or equal to *N*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *I* derived as follows:

a) For *j* varying from 1 (one) to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

b) For *j* varying from *M*+1 to *I*, the *j*-th element in *A* is the null value.

c) The *I*-th element of *A* is set to the value of *SV*, denoted by SV_i, by applying the General Rules of Subclause 9.2, "Store assignment", to the *I*-th element of *A* and SV_i as *TARGET* and *VALUE*, respectively.

3) Otherwise, an exception condition is raised: *data exception — array element error*.

2) Otherwise, the General Rules of Subclause 9.2, "Store assignment", are applied to TV_i and SV_i as *TARGET* and *VALUE*, respectively.

ii) If *TV* is a <host parameter name>, then the General Rules of Subclause 9.1, "Retrieval assignment", are applied to *TV* and *SV* as *TARGET* and *VALUE*, respectively.

iii) If *TV* is an <embedded variable specification>, then the General Rules of Subclause 9.1, "Retrieval assignment", are applied to *TV* and *SV* as *TARGET* and *VALUE*, respectively.

NOTE 380 — SQL parameters cannot have as their data types any row type.

6) If an exception condition occurs during the assignment of a value to a target, then the values of all targets are implementation-dependent.

NOTE 381 — It is implementation-dependent whether *CR* remains positioned on the current row when an exception condition is raised during the derivation of any <derived column>.

Conformance Rules

1) Without Feature F431, "Read-only scrollable cursors", a <fetch statement> shall not contain a <fetch orientation>.

14.6 <close statement>

This Subclause is modified by Subclause 12.4, "<close statement>", in ISO/IEC 9075-4.

Function

Close a standing cursor.

Format

```
<close statement> ::=
  CLOSE <cursor name>
```

Syntax Rules

1) Let *CN* be the <cursor name> in the <close statement>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

Access Rules

None.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) The General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

Conformance Rules

None.

14.7 <select statement: single row>

This Subclause is modified by Subclause 12.5, "<select statement: single row>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.14, "<select statement: single row>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 14.2, "<select statement: single row>", in ISO/IEC 9075-14.

Function

Retrieve values from a specified row of a table.

Format

```
<select statement: single row> ::=
  SELECT [ <set quantifier> ] <select list>
      INTO <select target list>
      <table expression>

<select target list> ::=
  <target specification> [ { <comma> <target specification> }... ]
```

Syntax Rules

1) <select target list> shall not contain a <target specification> that specifies a <column reference>.

2) Let *T* be the table defined by the <table expression>.

3) Case:

   a) If the <select target list> contains a single <target specification> *TS* and the degree of table *T* is greater than 1 (one), then the declared type of *TS* shall be a row type.

      Case:

      i) If *TS* is an <SQL parameter reference>, then the Syntax Rules of Subclause 9.2, "Store assignment", apply to *TS* and the row type of table *T* as *TARGET* and *VALUE*, respectively.

      ii) Otherwise, the Syntax Rules of Subclause 9.1, "Retrieval assignment", apply to *TS* and an arbitrary value of the row type of table *T* as *TARGET* and *VALUE*, respectively.

   b) Otherwise:

      i) The number of elements *NOE* in the <select list> shall be the same as the number of elements in the <select target list>. The *i*-th <target specification>, 1 (one) $\leq i \leq$ *NOE*, in the <select target list> corresponds with the *i*-th element of the <select list>.

      ii) For *i* varying from 1 (one) to *NOE*, let TS_i be the *i*-th <target specification> in the <select target list> that is either an <SQL parameter reference> or a <target array element specification>, and let SL_i be the *i*-th element of the <select list> that corresponds to the <target specification> in the <select target list>.

      Case:

1) If <target array element specification> is specified, then the Syntax Rules of Subclause 9.2, "Store assignment", apply to an arbitrary site whose declared type is the declared type of TS_i and SL_i as *TARGET* and *VALUE*, respectively.

2) Otherwise, the Syntax Rules of Subclause 9.2, "Store assignment", apply to TS_i and the corresponding element of the <select list>, as *TARGET* and *VALUE*, respectively.

iii) For each <target specification> *TS* that is a <host parameter specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", apply to *TS* and the corresponding element of the <select list>, as *TARGET* and *VALUE*, respectively.

iv) For each <target specification> *TS* that is an <embedded variable specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", apply to *TS* and the corresponding element of the <select list>, as *TARGET* and *VALUE*, respectively.

4) Let *S* be a <query specification> whose <select list> and <table expression> are those specified in the <select statement: single row> and that specifies the <set quantifier> if it is specified in the <select statement: single row>. *S* shall be a valid <query specification>.

5) A column in the result of the <select statement: single row> is known not null if the corresponding column in the result of *S* is known not null.

6) The <select statement: single row> is *possibly non-deterministic* if *S* is possibly non-deterministic.

Access Rules

None.

General Rules

1) Let *Q* be the result of <query specification> *S*.

2) Case:

a) If the cardinality of *Q* is greater than 1 (one), then an exception condition is raised: *cardinality violation*. It is implementation-dependent whether or not SQL-data values are assigned to the targets identified by the <select target list>.

b) If *Q* is empty, then no SQL-data values are assigned to any targets identified by the <select target list>, and a completion condition is raised: *no data*.

c) Otherwise, values in the row of *Q* are assigned to their corresponding targets.

3) If a completion condition *no data* has been raised, then no further General Rules of this Subclause are applied.

4) Case:

a) If the <select target list> contains a single <target specification> *TS* and the degree of table *T* is greater than 1 (one), then the current row is assigned to *TS* and

Case:

i) If *TS* is an <SQL parameter reference>, then the General Rules of Subclause 9.2, "Store assignment", apply to *TS* and the current row as *TARGET* and *VALUE*, respectively.

ii) Otherwise, the General Rules of Subclause 9.1, "Retrieval assignment", are applied to *TS* and the current row as *TARGET* and *VALUE*, respectively.

b) Otherwise:

i) Let *NOE* be the number of elements in the <select list>.

ii) For *i* varying from 1 (one) to *NOE*, let TS_i be the *i*-th <target specification> in the <select target list> that is either an <SQL parameter reference> or a <target array element specification>, and let SL_i denote the corresponding (*i*-th) value in the row of *Q*. The assignment of values to targets in the <select target list> is in an implementation-dependent order.

Case:

1) If <target array element specification> is specified, then

Case:

A) If the value of TS_i, denoted by *C*, is the null value, then an exception condition is raised: *data exception — null value in array target*.

B) Otherwise:

I) Let *N* be the maximum cardinality of *C*.

II) Let *M* be the cardinality of the value of *C*.

III) Let *I* be the value of the <simple value specification> immediately contained in TS_i.

IV) Let *EDT* be the element type of *C*.

V) Case:

1) If *I* is greater than zero and less than or equal to *M*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *M* derived as follows:

a) For *j* varying from 1 (one) to *I*–1 and from *I*+1 to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

b) The *I*-th element of *A* is set to the value of *SL*, denoted by SL_i, by applying the General Rules of Subclause 9.2, "Store assignment", to the *I*-th element of *A* and SL_i as *TARGET* and *VALUE*, respectively.

2) If *I* is greater than *M* and less than or equal to *N*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *I* derived as follows:

a) For *j* varying from 1 (one) to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

b) For *j* varying from *M*+1 to *I*, the *j*-th element in *A* is the null value.

c) The *I*-th element of *A* is set to the value of *SL*, denoted by SL_i, by applying the General Rules of Subclause 9.2, "Store assignment", to the *I*-th element of *A* and SL_i as *TARGET* and *VALUE*, respectively.

3) Otherwise, an exception condition is raised: *data exception — array element error*.

2) Otherwise, the corresponding value SL_i in the row of *Q* is assigned to TS_i according to the General Rules of Subclause 9.2, "Store assignment", as *VALUE* and *TARGET*, respectively.

iii) For each <target specification> *TS* that is a <host parameter specification>, the corresponding value in the row of *Q* is assigned to *TS* according to the General Rules of Subclause 9.1, "Retrieval assignment", as *VALUE* and *TARGET*, respectively. The assignment of values to targets in the <select target list> is in an implementation-dependent order.

iv) For each <target specification> *TS* that is an <embedded variable specification>, the corresponding value in the row of *Q* is assigned to *TS* according to the General Rules of Subclause 9.1, "Retrieval assignment", as *VALUE* and *TARGET*, respectively. The assignment of values to targets in the <select target list> is in an implementation-dependent order.

5) If an exception condition is raised during the assignment of a value to a target, then the values of all targets are implementation-dependent.

Conformance Rules

None.

14.8 <delete statement: positioned>

This Subclause is modified by Subclause 12.6, "<delete statement: positioned>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.12, "<delete statement: positioned>", in ISO/IEC 9075-10.

Function

Delete a row of a table.

Format

```
<delete statement: positioned> ::=
  DELETE FROM <target table> [ [ AS ] <correlation name> ]
      WHERE CURRENT OF <cursor name>

<target table> ::=
    <table name>
  | ONLY <left paren> <table name> <right paren>
```

Syntax Rules

1) Let *CN* be the <cursor name> in the <delete statement: positioned>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

4) The cursor specification of *CDD* shall be updatable.

5) Let *TN* be the <table name> contained in <target table>.

6) If <target table> *TT* immediately contains ONLY and the table identified by *TN* is not a typed table, then *TT* is equivalent to *TN*.

7) Let *T* be the simply underlying table of *CDD*. *T* is the *subject table* of the <delete statement: positioned>. Let *LUT* be the leaf underlying table of *T* such that *T* is one-to-one with respect to *LUT*.

8) The subject table of a <delete statement: positioned> shall not identify an old transition table or a new transition table.

9) *TN* shall identify *LUT*.

10) <target table> shall specify ONLY if and only if the <table reference> contained in *T* that references *LUT* specifies ONLY.

11) The schema identified by the explicit or implicit qualifier of *TN* shall include the descriptor of *LUT*.

12) Case:

    a) If <correlation name> is specified, then let *CN* be that <correlation name>.

b) Otherwise, let *CN* be the <table name> contained in <target table>. *CN* is an exposed <table or query name>.

NOTE 382 — *CN* has no scope.

Access Rules

1) Case:

a) If <delete statement: positioned> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges for the owner of that schema shall include DELETE for *TN*.

b) Otherwise, the current privileges shall include DELETE for *TN*.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) The General Rules of Subclause 15.5, "Effect of a positioned delete", are applied with *CR* as *CURSOR*.

Conformance Rules

1) Without Feature S111, "ONLY in query expressions", conforming SQL language shall not contain a <target table> that contains ONLY.

14.9 <delete statement: searched>

This Subclause is modified by Subclause 14.3, "<delete statement: searched>", in ISO/IEC 9075-14.

Function

Delete rows of a table.

Format

```
<delete statement: searched> ::=
  DELETE FROM <target table> [ [ AS ] <correlation name> ]
      [ WHERE <search condition> ]
```

Syntax Rules

1) Let *TN* be the <table name> contained in the <target table>. Let *T* be the table identified by *TN*.

2) *T* shall be an updatable table.

3) If the <delete statement: searched> is contained in a <triggered SQL statement>, then the <search condition> shall not contain a <value specification> that specifies a parameter reference.

4) *T* is the *subject table* of the <delete statement: searched>.

5) *TN* shall not identify an old transition table or a new transition table.

6) Case:

   a) If <correlation name> is specified, then let *CN* be that <correlation name>.

   b) Otherwise, let *CN* be the <table name> contained in <target table>. *CN* is an exposed <table or query name>.

7) The scope of *CN* is <search condition>.

8) If WHERE <search condition> is not specified, then WHERE TRUE is implicit.

9) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

Access Rules

1) Case:

   a) If <delete statement: searched> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema.

      i) The applicable privileges for *A* shall include DELETE for *TN*.

ii) If <target table> immediately contains ONLY, then the applicable privileges for *A* shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

b) Otherwise,

i) The current privileges shall include DELETE for *TN*.

ii) If <target table> immediately contains ONLY, then the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

General Rules

1) If the access mode of the current SQL-transaction or the access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

Case:

a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed*.

b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined.

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

5) The <search condition> is effectively evaluated for each row of *T* with the exposed <correlation name>s or <table or query name>s of the <table reference> bound to that row.

6) Case:

a) If <target table> contains ONLY, then the rows for which the result of the <search condition> is *True* and for which there is no subrow in a proper subtable of *T* are identified for deletion from *T*.

b) Otherwise, the rows for which the result of the <search condition> is *True* are identified for deletion from *T*.

NOTE 383 — Identifying a row for deletion is an implementation-dependent mechanism.

7) Case:

a) If *T* is a base table, then:

i) Case:

1) If <target table> specifies ONLY, then *T* is *identified for deletion processing without subtables*.

2) Otherwise, *T* is *identified for deletion processing with subtables*.

NOTE 384 — Identifying a base table for deletion processing, with or without subtables, is an implementation-dependent mechanism.

ii) The General Rules of Subclause 15.7, "Effect of deleting rows from base tables", are applied.

b) If *T* is a viewed table, then the General Rules of Subclause 15.9, "Effect of deleting some rows from a viewed table", are applied with <target table> as *VIEW NAME*.

8) If any row that is marked for deletion by the <delete statement: searched> has been marked for deletion by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, then a completion condition is raised: *warning — cursor operation conflict*.

9) If no rows are marked for deletion, then a completion condition is raised: *no data*.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <delete statement: searched> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in the <search condition>.

2) Without Feature T111, "Updatable joins, unions, and columns", conforming SQL language shall not contain a <delete statement: searched> that contains a <target table> that identifies a table that is not simply updatable.

14.10 <insert statement>

This Subclause is modified by Subclause 14.4, "<insert statement>", in ISO/IEC 9075-14.

Function

Create new rows in a table.

Format

```
<insert statement> ::=
  INSERT INTO <insertion target> <insert columns and source>

<insertion target> ::=
  <table name>

<insert columns and source> ::=
    <from subquery>
  | <from constructor>
  | <from default>

<from subquery> ::=
  [ <left paren> <insert column list> <right paren> ]
      [ <override clause> ]
      <query expression>

<from constructor> ::=
  [ <left paren> <insert column list> <right paren> ]
      [ <override clause> ]
      <contextually typed table value constructor>

<override clause> ::=
    OVERRIDING USER VALUE
  | OVERRIDING SYSTEM VALUE

<from default> ::=
  DEFAULT VALUES

<insert column list> ::=
  <column name list>
```

Syntax Rules

1) Let *TN* be the <table name>; let *T* be the table identified by *TN*. If *T* is a view, then <target table> is effectively replaced by:

   ```
   ONLY ( TN )
   ```

2) *T* shall be insertable-into.

3) For each leaf generally underlying table of *T* whose descriptor includes a user-defined type name *UDTN*, the data type descriptor of the user-defined type *UDT* identified by *UDTN* shall indicate that *UDT* is instantiable.

4) A column identified by the <insert column list> is an object column.

5) *T* shall be an updatable table; each object column of *T* shall be an updatable column.

   NOTE 385 — The notion of updatable columns of base tables is defined in Subclause 4.14, "Tables". The notion of updatable columns of viewed tables is defined in Subclause 11.22, "<view definition>".

6) *T* is the *subject table* of the <insert statement>.

7) *TN* shall not identify an old transition table or a new transition table.

8) An <insert columns and source> that specifies DEFAULT VALUES is implicitly replaced by an <insert columns and source> that specifies a <contextually typed table value constructor> of the form

   ```
   VALUES (DEFAULT, DEFAULT, ..., DEFAULT)
   ```

   where the number of "DEFAULT" entries is equal to the number of columns of *T*.

9) Each <column name> in the <insert column list> shall identify an updatable column of *T*. No <column name> of *T* shall be identified more than once. If the <insert column list> is omitted, then an <insert column list> that identifies all columns of *T* in the ascending sequence of their ordinal positions within *T* is implicit.

10) If <contextually typed table value constructor> *CTTVC* is specified, then every <contextually typed row value constructor element> simply contained in *CTTV*C whose positionally corresponding <column name> in <insert column list> references a column of which some underlying column is a generated column shall be a <default specification>.

11) Case:

    a) If some underlying column of a column referenced by a <column name> contained in <insert column list> is a system-generated self-referencing column or a derived self-referencing column, then <override clause> shall be specified.

    b) If for some *n*, some underlying column of the column referenced by the <column name> *CN* contained in the *n*-th ordinal position in <insert column list> is an identity column whose descriptor includes an indication that values are always generated, then

       Case:

       i) If <from subquery> is specified, then <override clause> shall be specified.

       ii) If any <contextually typed row value expression> simply contained in the <contextually typed table value constructor> is a <row value special case>, then <override clause> shall be specified.

       iii) If the *n*-th <contextually typed row value constructor element> simply contained in any <contextually typed row value constructor> simply contained in the <contextually typed table value constructor> is not a <default specification>, then <override clause> shall be specified.

       NOTE 386 — The preceding subrules do not cover all possibilities of their parent subrule. The remaining possibilities are where <default clause> is specified for every identity column, in which case it is immaterial whether <override clause> is specified or not.

    c) If for some *n*, some underlying column of the column referenced by the <column name> *CN* contained in the *n*-th ordinal position in <insert column list> is an identity column whose descriptor includes an indication that values are generated by default, then if <override clause> is specified, then <override clause> shall specify OVERRIDING USER VALUE.

    d) Otherwise, <override clause> shall not be specified.

12) If <contextually typed table value constructor> *CVC* is specified, then the data type of every <contextually typed value specification> *CVS* specified in every <contextually typed row value expression> *CRVS* contained in *CVC* is the data type *DT* indicated in the column descriptor for the positionally corresponding column in the explicit or implicit <insert column list>. If *CVS* is an <empty specification> that specifies ARRAY, then *DT* shall be an array type. If *CVS* is an <empty specification> that specifies MULTISET, then *DT* shall be a multiset type.

13) Let *QT* be the table specified by the <query expression> or <contextually typed table value constructor>. The degree of *QT* shall be equal to the number of <column name>s in the <insert column list>. The column of table *T* identified by the *i*-th <column name> in the <insert column list> corresponds with the *i*-th column of *QT*.

14) The Syntax Rules of Subclause 9.2, "Store assignment", apply to corresponding columns of *T* and *QT* as *TARGET* and *VALUE*, respectively.

15) If the <insert statement> is contained in a <triggered SQL statement>, then the insert value shall not contain a <value specification> that specifies a parameter reference.

16) A <query expression> simply contained in a <from subquery> shall not be a <table value constructor>.

NOTE 387 — This rule removes a syntactic ambiguity; otherwise, "VALUES (1)" could be parsed either as

```
<insert columns and source> ::=
   <from subquery> ::=
   <query expression> ::=
   <table value constructor> ::=
   VALUES (1)
```

or

```
<insert columns and source> ::=
   <from constructor> ::=
   <contextually typed table value constructor> ::=
   VALUES (1)
```

Access Rules

1) Case:

   a) If <insert statement> is contained in, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* for *TN* shall include INSERT for each object column.

   b) Otherwise, the current privileges for *TN* shall include INSERT for each object column.

General Rules

1) If the access mode of the current SQL-transaction or the access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

Case:

a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed*.

b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined.

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

5) *QT* is effectively evaluated before insertion of any rows into *T*.

6) Let *Q* be the result of evaluating *QT*.

7) For each row *R* of *Q*:

a) A candidate row of *T* is effectively created in which the value of each column is its default value, as specified in the General Rules of Subclause 11.5, "<default clause>". The candidate row consists of every column of *T*.

b) If *T* has a column *RC* of which some underlying column is a self-referencing column, then

Case:

i) If *RC* is a system-generated self-referencing column, then the value of *RC* is effectively replaced by the REF value of the candidate row.

ii) If *RC* is a derived self-referencing column, then the value of *RC* is effectively replaced by a value derived from the columns in the candidate row that correspond to the list of attributes of the derived representation of the reference type of *RC* in an implementation-dependent manner.

c) For each object column in the candidate row, let C_i be the object column identified by the *i*-th <column name> in the <insert column list> and let SV_i be the *i*-th value of *R*.

d) For every C_i for which one of the following conditions is true:

i) C_i is not marked as unassigned and no underlying column of C_i is a self-referencing column.

ii) Some underlying column of C_i is a user-generated self-referencing column.

iii) Some underlying column of C_i is a self-referencing column and OVERRIDING SYSTEM VALUE is specified.

iv) Some underlying column of C_i is an identity column and the *i*-th column of *R* is not derived from <default specification> and OVERRIDING SYSTEM VALUE is specified.

v) Some underlying column of C_i is an identity column whose descriptor includes an indication that values are generated by default and neither OVERRIDING USER VALUE is specified nor is the *i*-th column derived from <default specification>.

the General Rules of Subclause 9.2, "Store assignment", are applied with C_i and SV_i as *TARGET* and *SOURCE*, respectively. C_i is no longer marked as unassigned.

NOTE 388 — The data values allowable in the candidate row may be constrained by a WITH CHECK OPTION constraint. The effect of a WITH CHECK OPTION constraint is defined in the General Rules of Subclause 15.12, "Effect of inserting a table into a viewed table".

8) Let *S* be the table consisting of the candidate rows.

Case:

a) If *T* is a base table, then:

i) *T* is *identified for insertion of source table S*.

NOTE 389 — Identifying a base table for insertion of a source table is an implementation-dependent operation.

ii) The General Rules of Subclause 15.10, "Effect of inserting tables into base tables", are applied.

b) If *T* is a viewed table, then the General Rules of Subclause 15.12, "Effect of inserting a table into a viewed table", are applied with *S* as *SOURCE* and *T* as *TARGET*.

9) If *Q* is empty, then a completion condition is raised: *no data*.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain an <insert statement> in which the <table name> of a leaf generally underlying table of *T* is generally contained in the <from subquery> except as the table name of a qualifying table of a column reference.

2) Without Feature F222, "INSERT statement: DEFAULT VALUES clause", conforming SQL language shall not contain a <from default>.

3) Without Feature S024, "Enhanced structured types", in conforming SQL language, for each column *C* identified in the explicit or implicit <insert column list>, if the declared type of *C* is a structured type *TY*, then the declared type of the corresponding column of the <query expression> or <contextually typed table value constructor> shall be *TY*.

4) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain an <override clause>.

5) Without Feature T111, "Updatable joins, unions, and columns", conforming SQL language shall not contain an <insert statement> that contains an <insertion target> that identifies a table that is not simply updatable.

14.11 <merge statement>

This Subclause is modified by Subclause 14.5, "<merge statement>", in ISO/IEC 9075-14.

Function

Conditionally update rows of a table, or insert new rows into a table, or both.

Format

```
<merge statement> ::=
  MERGE INTO <target table> [ [ AS ] <merge correlation name> ]
      USING <table reference>
      ON <search condition> <merge operation specification>

<merge correlation name> ::=
  <correlation name>

<merge operation specification> ::=
  <merge when clause>...

<merge when clause> ::=
    <merge when matched clause>
  | <merge when not matched clause>

<merge when matched clause> ::=
  WHEN MATCHED THEN <merge update specification>

<merge when not matched clause> ::=
  WHEN NOT MATCHED THEN <merge insert specification>

<merge update specification> ::=
  UPDATE SET <set clause list>

<merge insert specification> ::=
  INSERT [ <left paren> <insert column list> <right paren> ]
      [ <override clause> ]
      VALUES <merge insert value list>

<merge insert value list> ::=
  <left paren>
      <merge insert value element> [ { <comma> <merge insert value element> }... ]
      <right paren>

<merge insert value element> ::=
    <value expression>
  | <contextually typed value specification>
```

Syntax Rules

1) Neither <merge when matched clause> nor <merge when not matched clause> shall be specified more than once.

2) Let *TN* be the <table name> contained in <target table> and let *T* be the table identified by *TN*. *T* is the *subject table* of the <merge statement>.

3) *T* shall be insertable-into.

4) *T* shall not be an old transition table or a new transition table.

5) For each leaf generally underlying table of *T* whose descriptor includes a user-defined type name *UDTN*, the data type descriptor of the user-defined type *UDT* identified by *UDTN* shall indicate that *UDT* is instantiable.

6) If *T* is a view, then <target table> is effectively replaced by:

```
ONLY ( TN )
```

7) Case:

   a) If <merge correlation name> is specified, then let *CN* be the <correlation name> contained in <merge correlation name>.

   b) Otherwise, let *CN* be the <table name> contained in <target table>.

8) The scope of *CN* is <search condition> and <set clause list>.

9) Let *TR* be the <table reference> immediately contained in <merge statement>. *TR* shall not directly contain a <joined table>.

10) The <correlation name> or exposed <table name> that is exposed by *TR* shall not be equivalent to *CN*.

11) If the <insert column list> is omitted, then an <insert column list> that identifies all columns of *T* in the ascending sequence of their ordinal position within *T* is implicit.

12) Case:

   a) If some underlying column of a column referenced by a <column name> contained in <insert column list> is a system-generated self-referencing column or a derived self-referencing column, then <override clause> shall be specified.

   b) If for some *n*, some underlying column of the column referenced by the <column name> *CN* contained in the *n*-th ordinal position in <insert column list> is an identity column whose descriptor includes an indication that values are always generated, then

      Case:

      i) If <from subquery> is specified, then <override clause> shall be specified.

      ii) If any <contextually typed row value expression> simply contained in the <contextually typed table value constructor> is a <row value special case>, then <override clause> shall be specified.

      iii) If the *n*-th <contextually typed row value constructor element> simply contained in any <contextually typed row value constructor> simply contained in the <contextually typed table value constructor> is not a <default specification>, then <override clause> shall be specified.

   c) If for some *n*, some underlying column of the column referenced by the <column name> *CN* contained in the *n*-th ordinal position in <insert column list> is an identity column whose descriptor includes an indication that values are generated by default, then if <override clause> is specified, then <override clause> shall specify OVERRIDING USER VALUE.

d) Otherwise, <override clause> shall not be specified.

13) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

14) Each column identified by an <object column> in the <set clause list> is an *update object column*. Each column identified by a <column name> in the implicit or explicit <insert column list> is an *insert object column*. Each update object column and each insert object column is an *object column*.

15) Every object column shall identify an updatable column of *T*.

NOTE 390 — The notion of updatable columns of base tables is defined in Subclause 4.14, "Tables". The notion of updatable columns of viewed tables is defined in Subclause 11.22, "<view definition>".

16) No <column name> of *T* shall be identified more than once in in an <insert column list>.

17) Let *NI* be the number of <merge insert value element>s contained in <merge insert value list>. Let EXP_1, EXP_2, ... , EXP_{NI} be those <merge insert value element>s.

18) The number of <column name>s in the <insert column list> shall be equal to *NI*.

19) The declared type of every <contextually typed value specification> *CVS* in a <merge insert value list> is the data type *DT* indicated in the column descriptor for the positionally corresponding column in the explicit or implicit <insert column list>. If *CVS* is an <empty specification> that specifies ARRAY, then *DT* shall be an array type. If *CVS* is an <empty specification> that specifies MULTISET, then *DT* shall be a multiset type.

20) Every <merge insert value element> whose positionally corresponding <column name> in <insert column list> references a column of which some underlying column is a generated column shall be a <default specification>.

21) For 1 (one) ≤ *i NI*, the Syntax Rules of Subclause 9.2, "Store assignment", apply to the column of table *T* identified by the *i*-th <column name> in the <insert column list> and EXP_i as *TARGET* and *VALUE*, respectively.

Access Rules

1) Case:

a) If <merge statement> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema.

i) The applicable privileges for *A* shall include UPDATE for each update object column.

ii) The applicable privileges for *A* shall include INSERT for each insert object column.

iii) If <target table> immediately contains ONLY, then the applicable privileges for *A* shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

b) Otherwise,

i) The current privileges shall include UPDATE for each update object column.

ii) The current privileges shall include INSERT for each insert object column.

iii) If <target table> immediately contains ONLY, then the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

General Rules

1) If the access mode of the current SQL-transaction or the access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

   Case:

   a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed*.

   b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined.

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

5) Let *QT* be the table specified by the <table reference>. *QT* is effectively evaluated before update or insertion of any rows in *T*. Let *Q* be the result of evaluating *QT*.

6) For each <merge when clause>,

   Case:

   a) If <merge when matched clause> is specified, then:

      i) For each row *R1* of *T*:

         1) The <search condition> is effectively evaluated for *R1* with the exposed <table name> of the <target table> bound to *R1* and to each row of *Q* with the exposed <correlation name>s or <table or query name>s of the <table reference> bound to that row. The <search condition> is effectively evaluated for *R1* before updating any row of *T* and prior to the invocation of any <triggered action> caused by the update of any row of *T* and before inserting any rows into *T* and prior to the invocation of any <triggered action> caused by the insert of any row of *T*.

            Case:

            A) If <target table> contains ONLY, then *R1* is a subject row if *R1* has no subrow in a proper subtable of *T* and the result of the <search condition> is <u>*True*</u> for some row *R2* of *Q*. *R2* is the matching row.

            B) Otherwise, *R1* is a subject row if the result of the <search condition> is <u>*True*</u> for some row *R2* of *Q*. *R2* is the matching row.

NOTE 391 — "outer reference" is defined in Subclause 6.7, "<column reference>".

2) If *R1* is a subject row, then:

A) Let *M* be the number of matching rows in *Q* for *R1*.

B) If *M* is greater than 1 (one), then an exception condition is raised: *cardinality violation*.

C) The <update source> of each <set clause> is effectively evaluated for *R1* before any row of *T* is updated and prior to the invocation of any <triggered action> caused by the update of any row of *T*. The resulting value is the update value.

D) A candidate new row is constructed by copying the subject row and updating it as specified by each <set clause> by applying the General Rules of Subclause 14.14, "<set clause list>".

ii) If *T* is a base table, then each subject row is also an object row; otherwise, an object row is any row of a leaf generally underlying table of *T* from which a subject row is derived.

NOTE 392 — The data values allowable in the object rows may be constrained by a WITH CHECK OPTION constraint. The effect of a WITH CHECK OPTION constraint is defined in the General Rules of Subclause 15.15, "Effect of replacing some rows in a viewed table".

iii) If any row in the set of object rows has been marked for deletion by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, then a completion condition is raised: *warning — cursor operation conflict*.

iv) Let *CL* be the columns of *T* identified by the <object column>s contained in the <set clause list>.

v) Each subject row *SR* is identified for replacement, by its corresponding candidate new row *CNR*, in *T*. The set of (*SR*, *CNR*) pairs is the replacement set for *T*.

NOTE 393 — Identifying a row for replacement, associating a replacement row with an identified row, and associating a replacement set with a table are implementation-dependent operations.

vi) Case:

1) If *T* is a base table, then:

A) Case:

I) If <target table> specifies ONLY, then *T* is *identified for replacement processing without subtables* with respect to object columns *CL*.

II) Otherwise, *T* is *identified for replacement processing with subtables* with respect to object columns *CL*.

NOTE 394 — Identifying a base table for replacement processing, with or without subtables, is an implementation-dependent mechanism. In general, though not here, the list of object columns can be empty.

B) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

2) If *T* is a viewed table, then the General Rules of Subclause 15.15, "Effect of replacing some rows in a viewed table", are applied with <target table> as *VIEW NAME*.

b) If <merge when not matched clause> is specified, then:

i) Let *TR1* be the <target table> immediately contained in <merge statement>, let *TR2* be the <table reference> immediately contained in <merge statement>, and let *SC1* be the <search condition> immediately contained in <merge statement>. If <merge correlation name> is specified, let *MCN* be "AS <merge correlation name>"; otherwise, let *MCN* be a zero-length string. Let *S1* be the result of

```
SELECT *
FROM TR1 MCN, TR2
WHERE SC1
```

ii) Let *S2* be the collection of rows of *Q* for which there exists in *S1* some row that is the concatenation of some row *R1* of *T* and some row *R2* of *Q*.

iii) Let *S3* be the collection of rows of *Q* that are not in *S2*. Let *SN3* be the effective distinct name for *S3*. Let *EN* be the exposed <correlation name> or <table or query name> of *TR2*.

iv) Let *S4* be the result of:

```
SELECT EXP1, EXP2, ... , EXPNI
FROM SN3 AS EN
```

v) *S4* is effectively evaluated before insertion of any rows into or update of any rows in *T*.

vi) For each row *R* of *S4*:

1) A candidate row of *T* is effectively created in which the value of each column is its default value, as specified in the General Rules of Subclause 11.5, "<default clause>". The candidate row consists of every column of *T*.

2) If *T* has a column *RC* of which some underlying column is a self-referencing column, then

Case:

A) If *RC* is a system-generated self-referencing column, then the value of *RC* is effectively replaced by the REF value of the candidate row.

B) If *RC* is a derived self-referencing column, then the value of *RC* is effectively replaced by a value derived from the columns in the candidate row that correspond to the list of attributes of the derived representation of the reference type of *RC* in an implementation-dependent manner.

3) For each object column in the candidate row, let C_i be the object column identified by the *i*-th <column name> in the <insert column list> and let SV_i be the *i*-th value of *R*.

4) For every C_i for which one of the following conditions is true:

A) C_i is not marked as unassigned and no underlying column of C_i is a self-referencing column.

B) Some underlying column of C_i is a user-generated self-referencing column.

C) Some underlying column of C_i is a self-referencing column and OVERRIDING SYSTEM VALUE is specified.

D) Some underlying column of C_i is an identity column and the *i*-th column of *R* is not derived from <default specification> and OVERRIDING SYSTEM VALUE is specified.

E) Some underlying column of C_i is an identity column whose descriptor includes an indication that values are generated by default and neither OVERRIDING USER VALUE is specified nor is the *i*-th column derived from <default specification>.

the General Rules of Subclause 9.2, "Store assignment", are applied to C_i and SV_i as *TARGET* and *SOURCE*, respectively. C_i is no longer marked as unassigned.

NOTE 395 — The data values allowable in the candidate row may be constrained by a WITH CHECK OPTION constraint. The effect of a WITH CHECK OPTION constraint is defined in the General Rules of Subclause 15.12, "Effect of inserting a table into a viewed table".

vii) Let *S* be the table consisting of the candidate rows.

Case:

1) If *T* is a base table, then:

A) *T* is *identified for insertion of source table S*.

NOTE 396 — Identifying a base table for insertion of a source table is an implementation-dependent operation.

B) The General Rules of Subclause 15.10, "Effect of inserting tables into base tables", are applied.

2) If *T* is a viewed table, then the General Rules of Subclause 15.12, "Effect of inserting a table into a viewed table", are applied with *S* as *SOURCE* and *T* as *TARGET*.

7) If *Q* is empty, then a completion condition is raised: *no data*.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <merge statement> in which a leaf generally underlying table of *T* is generally contained in a <query expression> immediately contained in the <table reference> except as the <table or query name> or <correlation name> of a column reference.

2) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <merge statement> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in the <search condition>.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <merge statement> that does not satisfy the condition: for each column *C* identified in the explicit or implicit <insert column list>, if the declared type of *C* is a structured type *TY*, then the declared type of the corresponding column of the <query expression> or <contextually typed table value constructor> is *TY*.

4) Without Feature F312, "MERGE statement", conforming SQL language shall not contain a <merge statement>.

5) Without Feature T111, "Updatable joins, unions, and columns", conforming SQL language shall not contain a <merge statement> that contains an <target table> that identifies a table that is not simply updatable.

14.12 <update statement: positioned>

This Subclause is modified by Subclause 12.7, "<update statement: positioned>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.13, "<update statement: positioned>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 14.6, "<update statement: positioned>", in ISO/IEC 9075-14.

Function

Update a row of a table.

Format

```
<update statement: positioned> ::=
  UPDATE <target table> [ [ AS ] <correlation name> ]
      SET <set clause list>
      WHERE CURRENT OF <cursor name>
```

Syntax Rules

1) Let *CN* be the <cursor name> in the <update statement: positioned>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

4) The cursor specification of *CDD* shall be updatable.

5) Let *TU* be the simply underlying table of *CDD*. *TU* is the *subject table* of the <update statement: positioned>. Let *LUT* be the leaf underlying table *T* such that *T* is one-to-one with respect to *LUT*.

6) Let *TN* be the <table name> contained in <target table>. *TN* shall identify *LUT*.

7) <target table> shall specify ONLY if and only if the <table reference> contained in *TU* that references *LUT* specifies ONLY.

8) *TN* shall not identify an old transition table or a new transition table.

9) Let *T* be the table identified by *TN*.

10) Case:

    a) If <correlation name> is specified, then let *CN* be that <correlation name>.

    b) Otherwise, let *CN* be the <table name> contained in <target table>. *CN* is an exposed <table or query name>.

11) The scope of *CN* is <set clause list>.

12) If the declared <cursor specification> of *CDD* is ordered, then for each <object column> *OC* contained in <set clause list>, the <order by clause> of the <cursor specification> of *CDD* shall not generally contain a <column reference> that references *OC* or an underlying column of the column identified by *OC*.

13) Each <column name> specified as an <object column> shall identify a column in the explicit or implicit <column name list> contained in the explicit or implicit <updatability clause> of the <cursor specification> of *CDD*.

Access Rules

1) Case:

   a) If <update statement: positioned> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* shall include UPDATE for each <object column>.

   b) Otherwise, the current privileges shall include UPDATE for each <object column>.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) Let *SCL* be the <set clause list>.

3) The General Rules of Subclause 15.6, "Effect of a positioned update", are applied, with *CR* as *CURSOR* and *SCL* as *SET CLAUSE LIST*.

Conformance Rules

1) Without Feature F831, "Full cursor update", conforming SQL language shall not contain an <update statement: positioned> in which the declared <cursor specification> of *CDD* is ordered.

14.13 <update statement: searched>

This Subclause is modified by Subclause 14.7, "<update statement: searched>", in ISO/IEC 9075-14.

Function

Update rows of a table.

Format

```
<update statement: searched> ::=
  UPDATE <target table> [ [ AS ] <correlation name> ]
      SET <set clause list>
      [ WHERE <search condition> ]
```

Syntax Rules

1) Let *TN* be the <table name> contained in <target table>; let *T* be the table identified by *TN*. *T* shall be an updatable table.

2) *T* is the *subject table* of the <update statement: searched>.

3) *TN* shall not identify an old transition table or a new transition table.

4) Case:

   a) If <correlation name> is specified, then let *CN* be that <correlation name>.

   b) Otherwise, let *CN* be the <table name> contained in <target table>. *CN* is an exposed <table or query name>.

5) The scope of *CN* is <set clause list> and <search condition>.

6) If the <update statement: searched> is contained in a <triggered SQL statement>, then the <search condition> shall not contain a <value specification> that specifies a parameter reference.

7) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

Access Rules

1) Case:

   a) If <update statement: searched> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema.

      i) The applicable privileges for *A* for *TN* shall include UPDATE for each <object column>.

      ii) If <target table> immediately contains ONLY, then the applicable privileges for *A* shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

b) Otherwise,

i) The current privileges for *TN* shall include UPDATE for each <object column>.

ii) If <target table> immediately contains ONLY, then the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

General Rules

1) If the access mode of the current SQL-transaction or the access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

Case:

a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed*.

b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined.

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

5) Case:

a) If <target table> contains ONLY, then

Case:

i) If a <search condition> is not specified, then all rows of *T* for which there is no subrow in a proper subtable of *T* are the subject rows.

ii) If a <search condition> is specified, then it is effectively evaluated for each row of *T* with the exposed <correlation name>s or <table or query name>s of the <table reference> bound to that row, and the subject rows are those rows for which the result of the <search condition> is *True* and for which there is no subrow in a proper subtable of *T*. The <search condition> is effectively evaluated for each row of *T* before updating any row of *T*.

b) Otherwise,

Case:

i) If a <search condition> is not specified, then all rows of *T* are the subject rows.

ii) If a <search condition> is specified, then it is effectively evaluated for each row of *T* with the exposed <table name> of the <target table> bound to that row, and the subject rows are those

rows for which the result of the <search condition> is *True*. The <search condition> is effectively evaluated for each row of *T* before any row of *T* is updated.

6) If *T* is a base table, then each subject row is also an *object row*; otherwise, an *object row* is any row of a leaf generally underlying table of *T* from which a subject row is derived.

7) If any row in the set of object rows has been marked for deletion by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, then a completion condition is raised: *warning — cursor operation conflict*.

8) If a <search condition> is specified, then the <search condition> is evaluated for each row of *T* prior to the invocation of any <triggered action> caused by the update of any row of *T*.

9) The <update source> of each <set clause> is effectively evaluated for each row of *T* before any row of *T* is updated.

10) For each subject row, a candidate new row is constructed by copying the subject row and updating it as specified by each <set clause> by applying the General Rules of Subclause 14.14, "<set clause list>".

    NOTE 397 — The data values allowable in the object rows may be constrained by a WITH CHECK OPTION constraint. The effect of a WITH CHECK OPTION constraint is defined in the General Rules of Subclause 15.15, "Effect of replacing some rows in a viewed table".

11) Let *CL* be the columns of *T* identified by the <object column>s contained in the <set clause list>.

12) Each subject row *SR* is identified for replacement, by its corresponding candidate new row *CNR*, in *T*. The set of `(SR, CNR)` pairs is the *replacement set* for *T*.

    NOTE 398 — Identifying a row for replacement, associating a replacement row with an identified row, and associating a replacement set with a table are implementation-dependent operations.

13) Case:

    a) If *T* is a base table, then:

       i) Case:

          1) If <target table> specifies ONLY, then *T* is *identified for replacement processing without subtables* with respect to object columns *CL*.

          2) Otherwise, *T* is *identified for replacement processing with subtables* with respect to object columns *CL*.

          NOTE 399 — Identifying a base table for replacement processing, with or without subtables, is an implementation-dependent mechanism. In general, though not here, the list of object columns can be empty.

       ii) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

    b) If *T* is a viewed table, then the General Rules of Subclause 15.15, "Effect of replacing some rows in a viewed table", are applied with <target table> as *VIEW NAME*.

14) If the set of object rows is empty, then a completion condition is raised: *no data*.

Conformance Rules

1) Without Feature F781, “Self-referencing operations”, conforming SQL language shall not contain an <update statement: searched> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in the <search condition>.

2) Without Feature T111, “Updatable joins, unions, and columns”, conforming SQL language shall not contain an <update statement: searched> that contains a <target table> that identifies a table that is not simply updatable.

14.14 <set clause list>

Function

Specify a list of updates.

Format

```
<set clause list> ::=
  <set clause> [ { <comma> <set clause> }... ]

<set clause> ::=
    <multiple column assignment>
  | <set target> <equals operator> <update source>

<set target> ::=
    <update target>
  | <mutated set clause>

<multiple column assignment> ::=
  <set target list> <equals operator> <assigned row>

<set target list> ::=
  <left paren> <set target> [ { <comma> <set target> }... ] <right paren>

<assigned row> ::=
  <contextually typed row value expression>

<update target> ::=
    <object column>
  | <object column>
      <left bracket or trigraph> <simple value specification> <right bracket or trigraph>

<object column> ::=
  <column name>

<mutated set clause> ::=
  <mutated target> <period> <method name>

<mutated target> ::=
    <object column>
  | <mutated set clause>

<update source> ::=
    <value expression>
  | <contextually typed value specification>
```

Syntax Rules

1) Let *T* be the table identified by the <target table> contained in the containing <update statement: positioned>, <update statement: searched>, or <merge statement>.

2) Each <column name> specified as an <object column> shall identify an updatable column of *T*.

NOTE 400 — The notion of updatable columns of base tables is defined in Subclause 4.14, "Tables". The notion of updatable columns of viewed tables is defined in Subclause 11.22, "<view definition>".

3) Each <set clause> *SC* that immediately contains a <multiple column assignment> is effectively replaced by a <set clause list> *MSCL* as follows:

   a) Let *STN* be the number of <set target>s contained in <set target list>.

   b) *STN* shall be equal to the degree of the <assigned row> *AR* contained in *SC*.

   c) Let ST_i, 1 (one) ≤ i ≤ *STN*, be the i-th <set target> contained in the <set target list> of *SC* and let DT_i be the declared type of the i-th field of *AR*. The i-th <set clause> in *MSCL* is:

```
ST_i =
CAST ( AR AS ROW ( F1 DT_1,
F2 DT_2, ...,
FSTN DT_STN ) ).Fi
```

   NOTE 401 — "F*n*" here stands for the <field name> consisting of the letter "F" followed, with no intervening <separator> by the decimal <digit> or <digit>s comprising a <literal> corresponding to the value *n*.

4) If <set clause> *SC* specifies an <object column> that references a column of which some underlying column is either a generated column or an identity column whose descriptor indicates that values are always generated, then the <update source> specified in *SC* shall consist of a <default specification>.

5) A <value expression> simply contained in an <update source> in a <set clause> shall not directly contain a <set function specification>.

6) If the <set clause list> *OSCL* contains one or more <set clause>s that contain a <mutated set clause>, then:

   a) Let *N* be the number of <set clause>s in *OSCL* that contain a <mutated set clause>.

   b) For 1 (one) ≤ i ≤ *N*:

      i) Let SC_i be the i-th <set clause> that contains a <mutated set clause>.

      ii) Let $RCVE_i$ be the <update source> immediately contained in SC_i.

      iii) Let MSC_i be the <mutated set clause> immediately contained in the <set target> immediately contained in SC_i.

      iv) Let OC_i be the <object column> contained in MSC_i. The declared type of the column identified by OC_i shall be a structured type.

      v) Let M_i be the number of <method name>s contained in MSC_i.

      vi) For 1 (one) ≤ j ≤ M_i:

         1) If j = 1 (one), then

            Case:

            A) Let $MT_{i,1}$ be the <mutated target> immediately contained in MSC_i.

            B) Let $MN_{i,1}$ be the <method name> immediately contained in MSC_i.

C) Let $V_{i,1}$ be:

$MT_{i,1}$. $MN_{i,1}$ ($RCVE_i$)

2) Otherwise:

A) Let $MT_{i,j}$ be the <mutated target> immediately contained in the <mutated set clause> immediately contained in $MT_{i,j-1}$.

B) Let $MN_{i,j}$ be the <method name> immediately contained in the <mutated set clause> immediately contained in $MT_{i,j-1}$.

C) Let $V_{i,j}$ be

$MT_{i,j}$. $MN_{i,j}$ ($V_{i,j-1}$)

c) *OSCL* is equivalent to a <set clause list> *NSCL* derived as follows:

i) Let *NSCL* be a <set clause list> derived from *OSCL* by replacing every <set clause> SC_a, 1 (one) $\leq a \leq N$, that contains a <mutated set clause> with:

$MT_{a,Ma} = V_{a,Ma}$

ii) For 1 (one) $\leq b \leq N$, if there exists a c such that $c < b$ and OC_c is equivalent to OC_b, then:

1) Every occurrence of OC_b in $V_{b,Mb}$ is replaced by $V_{c,Mc}$.

2) SC_c is deleted from *NSCL*.

7) Equivalent <object column>s shall not appear more than once in a <set clause list>.

NOTE 402 — Multiple occurrences of equivalent <object column>s within <mutated set clause>s are eliminated by the preceding Syntax Rule of this Subclause.

8) If the <update source> of <set clause> *SC* specifies a <contextually typed value specification> *CVS*, then the data type of *CVS* is the data type *DT* of the <update target> or <mutated set clause> specified in *SC*.

9) If *CVS* is an <empty specification>, then *DT* shall be a collection type. If *CVS* specifies ARRAY, then *DT* shall be an array type. If *CVS* specifies MULTISET, then *DT* shall be a multiset type.

10) For every <object column> in a <set clause>,

Case:

a) If the <update target> immediately contains <simple value specification>, then the declared type of the column of *T* identified by the <object column> shall be an array type. The Syntax Rules of Subclause 9.2, "Store assignment", apply to an arbitrary site whose declared type is the element type of the column of *T* identified by the <object column> and the <update source> of the <set clause> as *TARGET* and *VALUE*, respectively.

b) Otherwise, the Syntax Rules of Subclause 9.2, "Store assignment", apply to the column of *T* identified by the <object column> and the <update source> of the <set clause> as *TARGET* and *VALUE*, respectively.

Access Rules

None.

General Rules

1) A <set clause> specifies one or more object columns and an update value. An *object column* is a column identified by an <object column> in the <set clause>. The *update value* is the value specified by the <update source> contained in the <set clause>.

2) The value of the *i*-th object column denoted by *C*, is replaced as follows.

   Case:

   a) If the *i*-th <set clause> contains an <update target> that immediately contains a <simple value specification>, then

      Case:

      i) If the value of *C* is the null value, then an exception condition is raised: *data exception — null value in array target*.

      ii) Otherwise:

         1) Let *N* be the maximum cardinality of *C*.

         2) Let *M* be the cardinality of the value of *C*.

         3) Let *I* be the value of the <simple value specification> immediately contained in <update target>.

         4) Let *EDT* be the element type of *C*.

         5) Case:

            A) If *I* is greater than zero and less than or equal to *M*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *M* derived as follows:

               I) For *j* varying from 1 (one) to *I*–1 and from *I*+1 to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

               II) The *I*-th element of *A* is set to the *i*-th update value, denoted by *SV*, by applying the General Rules of Subclause 9.2, "Store assignment", to the *I*-th element of *A* and *SV* as *TARGET* and *VALUE*, respectively.

            B) If *I* is greater than *M* and less than or equal to *N*, then the value of *C* is replaced by an array *A* with element type *EDT* and cardinality *I* derived as follows:

               I) For *j* varying from 1 (one) to *M*, the *j*-th element in *A* is the value of the *j*-th element in *C*.

               II) For *j* varying from *M*+1 to *I*–1, the *j*-th element in *A* is the null value.

               III) The *I*-th element of *A* is set to the *i*-th update value, denoted by *SV*, by applying the General Rules of Subclause 9.2, "Store assignment", to the *I*-th element of *A* and *SV* as *TARGET* and *VALUE*, respectively.

C) Otherwise, an exception condition is raised: *data exception — array element error*.

b) Otherwise, the value of *C* is replaced by the *i*-th update value, denoted by *SV*. The General Rules of Subclause 9.2, "Store assignment", are applied to *C* and *SV* as *TARGET* and *VALUE*, respectively.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <set clause> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in any <value expression> simply contained in an <update source> or <assigned row> immediately contained in the <set clause>.

2) Without Feature S091, "Basic array support", conforming SQL language shall not contain an <update target> that immediately contains a <simple value specification>.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <set clause> in which the declared type of the <update target> in the <set clause> is a structured type *TY* and the declared type of the <update source> or corresponding field of the <assigned row> contained in the <set clause> is not *TY*.

4) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <set clause> that contains a <mutated set clause> and in which the declared type of the last <method name> identifies a structured type *TY*, and the declared type of the <update source> contained in the <set clause> is not *TY*.

5) Without Feature T641, "Multiple column assignment", conforming SQL language shall not contain a <multiple column assignment>.

14.15 <temporary table declaration>

This Subclause is modified by Subclause 12.8, "<temporary table declaration>", in ISO/IEC 9075-4.

Function

Declare a declared local temporary table.

Format

```
<temporary table declaration> ::=
  DECLARE LOCAL TEMPORARY TABLE <table name> <table element list>
      [ ON COMMIT <table commit action> ROWS ]
```

Syntax Rules

1) Let *TN* be the <table name> of a <temporary table declaration> *TTD*, and let *T* be the <qualified identifier> of *TN*.

2) *TTD* shall be contained in an <SQL-client module definition>.

3) Case:

   a) If *TN* contains a <local or schema qualifier> *LSQ*, then *LSQ* shall be "MODULE".

   b) If *TN* does not contain a <local or schema qualifier>, then "MODULE" is implicit.

4) If a <temporary table declaration> is contained in an <SQL-client module definition> *M*, then the <qualified identifier> of *TN* shall not be equivalent to the <qualified identifier> of the <table name> of any other <temporary table declaration> that is contained in *M*.

5) The descriptor of the table defined by a <temporary table declaration> includes *TN* and the column descriptor specified by each <column definition>. The *i*-th column descriptor is given by the *i*-th <column definition>.

6) A <temporary table declaration> shall contain at least one <column definition>.

7) If ON COMMIT is not specified, then ON COMMIT DELETE ROWS is implicit.

Access Rules

None.

General Rules

1) Let *U* be the implementation-dependent <schema name> that is effectively derived from the implementation-dependent SQL-session identifier associated with the SQL-session and an implementation-dependent name associated with the SQL-client module that contains the <temporary table declaration>.

2) Let *UI* be the current user identifier and let *R* be the current role name.

Case:

a) If *UI* is not the null value, then let *A* be *UI*.

b) Otherwise, let *A* be *R*.

3) The definition of *T* within an SQL-client module is effectively equivalent to the definition of a persistent base table *U.T*. Within the SQL-client module, any reference to MODULE.*T* is equivalent to a reference to *U.T*.

4) A set of privilege descriptors is created that define the privileges INSERT, SELECT, UPDATE, DELETE, and REFERENCES on this table and INSERT, SELECT, UPDATE, and REFERENCES for every <column definition> in the table definition to *A*. These privileges are not grantable. The grantor for each of these privilege descriptors is set to the special grantor value "_SYSTEM". The grantee is "PUBLIC".

5) The definition of a temporary table persists for the duration of the SQL-session. The termination of the SQL-session is effectively followed by the execution of the following <drop table statement> with the current authorization identifier *A* and current <schema name> *U* without further Access Rule checking:

```
DROP TABLE T CASCADE
```

6) The definition of a declared local temporary table does not appear in any view of the Information Schema.

NOTE 403 — The Information Schema is defined in ISO/IEC 9075-11.

Conformance Rules

1) Without Feature F531, "Temporary tables", conforming SQL language shall not contain a <temporary table declaration>.

14.16 <free locator statement>

Function

Remove the association between a locator variable and the value that is represented by that locator.

Format

```
<free locator statement> ::=
  FREE LOCATOR <locator reference> [ { <comma> <locator reference> }... ]

<locator reference> ::=
    <host parameter name>
  | <embedded variable name>
  | <dynamic parameter specification>
```

Syntax Rules

1) Each host parameter identified by <host parameter name> immediately contained in <locator reference> shall be a binary large object locator parameter, a character large object locator parameter, an array locator parameter, a multiset locator parameter, or a user-defined type locator parameter.

2) Each host variable identified by the <embedded variable name> immediately contained in <locator reference> shall be a binary object locator variable, a character large object locator variable, an array locator variable, a multiset locator parameter, or a user-defined type locator variable.

Access Rules

None.

General Rules

1) For every <locator reference> *LR* immediately contained in <free locator statement>, let *L* be the value of *LR*.

   Case:

   a) If *L* is not a valid locator value, then an exception condition is raised: *locator exception — invalid specification*.

   b) Otherwise, *L* is marked invalid.

Conformance Rules

1) Without Feature T561, “Holdable locators”, conforming SQL language shall not contain a <free locator statement>.

14.17 <hold locator statement>

Function

Mark a locator variable as being holdable.

Format

```
<hold locator statement> ::=
  HOLD LOCATOR <locator reference> [ { <comma> <locator reference> }... ]
```

Syntax Rules

1) Each host parameter identified by <host parameter name> immediately contained in <locator reference> shall be a binary large object locator parameter, a character large object locator parameter, an array locator parameter, a multiset locator parameter, or a user-defined type locator parameter.

Access Rules

None.

General Rules

1) For every <locator reference> *LR* immediately contained in <hold locator statement>, let *L* be the value of *LR*.

   Case:

   a) If *L* is not a valid locator value, then an exception condition is raised: *locator exception — invalid specification*.

   b) Otherwise, *L* is marked *holdable*.

Conformance Rules

1) Without Feature T561, “Holdable locators”, conforming SQL language shall not contain a <hold locator statement>.

(Blank page)

15 Additional data manipulation rules

This Clause is modified by Clause 15, "Additional data manipulation rules", in ISO/IEC 9075-9.

15.1 Effect of opening a cursor

Function

Specify the effect of opening a cursor that is not a received cursor.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* specified in an application of this Subclause. If *CR* is not in the closed state, then an exception condition is raised: *invalid cursor state*.

2) Let *CDD* be the cursor declaration descriptor of *CR*.

3) Let *S* be the declared <cursor specification> of *CDD*.

4) *CR* is opened, and a result set descriptor *RSD* is created and included in *CR*, in the following steps:

   a) The <cursor specification> of *RSD* is a copy *CS* of *S* that is effectively created as follows:

      i) Each <embedded variable specification>, <host parameter specification>, <SQL parameter reference>, and <dynamic parameter specification> is replaced by a <literal> denoting the value resulting from evaluating the <embedded variable specification>, <host parameter specification>, <SQL parameter reference>, and <dynamic parameter specification>, respectively, with all such evaluations effectively done at the same instant in time.

      ii) Each <value specification> generally contained in *S* that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_PATH, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is replaced by a <literal> denoting the value resulting from evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_PATH, CURRENT_DEFAULT_TRANSFORM_GROUP, or

CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>, respectively, with all such evaluations effectively done at the same instant in time.

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

b) Case:

i) If *CR* is a standing cursor, then the operational properties of *RSD* are the same as the corresponding declared properties of *CDD*.

ii) Otherwise, *CR* is a dynamic cursor.

1) The operational properties of *RSD* are initially copied from the corresponding declared properties of *CDD*.

2) If the <prepare statement> that prepared the <cursor specification> *S* contained an <attributes variable>, then let *CA* be the value of that <attributes variable>.

A) If *CA* contains a <cursor sensitivity>, then the operational sensitivity property of *RSD* is set to that <cursor sensitivity>.

B) If *CA* contains a <cursor scrollability>, then the operational scrollability property of *RSD* is set to that <cursor scrollability>.

C) If *CA* contains a <cursor holdability>, then the operational holdability property of *RSD* is set to that <cursor holdability>.

D) If *CA* contains a <cursor returnability>, then the operational returnability property of *RSD* is set to that <cursor returnability>.

c) If *CR* is being opened during either

i) the execution of an SQL-invoked function, or

ii) the execution of an externally-invoked procedure

without the intervening execution of an SQL-invoked procedure, then the operational returnability property of *RSD* is set to WITHOUT RETURN.

d) Let *T* be the table specified by *CS*.

e) Case:

i) If the operational sensitivity property of *RSS* is INSENSITIVE, then let *TT* be a copy of *T*.

ii) Otherwise, let *TT* be *T*.

f) The sequence of rows of *RSS* consists of the rows of *TT*, ordered as follows:

i) If an <order by clause> is not specified, then the sequence of rows specified by the <cursor specification> is *T* and the ordering of rows in *T* is implementation-dependent.

ii) If an <order by clause> is specified, then the ordering of rows of the result set is determined by the General Rules of Subclause 10.10, "<sort specification list>". The result set specified by the <cursor specification> is the sort table of *CS* with all extended sort key columns (if any) removed.

NOTE 404 — "extended set key column" is defined in Subclause 14.3, "<cursor specification>".

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

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

NOTE 405 — The visibility of significant changes through a sensitive holdable cursor during a subsequent SQL-transaction is implementation-defined.

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

8) If the operational returnability property of *RSD* is WITH RETURN, then let *SIP* be the active SQL-invoked procedure, let *INV* be the invoker of *SIP*, and let *RSS* be the result set sequence for *SIP* and *INV* in the active SQL-session context. *RSD* is added to the end of *RSS*.

Conformance Rules

None.

15.2 Effect of advancing a received cursor to the next result set

Function

Advance a received cursor to the next result set in a result set sequence.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* in an application of this Subclause.

2) Let *SIP* be the SQL-invoked procedure indicated by the <specific routine designator> included in the cursor declaration descriptor of *CR*.

3) Let *RSS* be the result set sequence associated with *SIP*.

4) Let *RS* be the first result sequence descriptor in *RSS*.

5) *CR* is placed in the open state, with *RS* as the result set descriptor included in *CR*.

6) *RS* is removed from *RSS*.

7) If the operational scrollability property of *RS* is NO SCROLL, then the position of *RS* is set to before the first row.

8) The operational sensitivity property of *RS* is set to ASENSITIVE.

9) The operational scrollability property of *RS* is set to NO SCROLL.

10) The operational holdability property of *RS* is set to WITHOUT HOLD.

11) The operational returnability property of *RS* is set to WITHOUT RETURN.

12) The <cursor specification> of *RS* is made not updatable by replacing the explicit or implicit <updatability clause> with FOR READ ONLY.

Conformance Rules

None.

15.3 Determination of the current row of a cursor

Function

Specify how the current row of a cursor is determined.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* and let *FO* be the *FETCH ORIENTATION* specified in an application of this Subclause.

2) If *CR* is not in the open state, then an exception condition is raised: *invalid cursor state*.

3) Let *RSD* be the result set descriptor of *CR*.

4) If *FO* is not NEXT and the operational scrollability property of *RSD* is not SCROLL, then an exception condition is raised: *syntax error or access rule violation*.

5) Case:

   a) If *FO* contains a <simple value specification>, then let *J* be the value of that <simple value specification>.

   b) If *FO* specifies NEXT or FIRST, then let *J* be +1.

   c) If *FO* specifies PRIOR or LAST, then let *J* be –1.

6) Let *T* be the sequence of rows of *RSD*.

7) Let T_t be a sequence of rows of the same degree as *T*.

   Case:

   a) If *FO* specifies ABSOLUTE, FIRST, or LAST, then let T_t contain all rows of *T*, preserving their order in *T*.

   b) If *FO* specifies NEXT or specifies RELATIVE with a positive value of *J*, then:

      i) If *T* is empty or if the position of *CR* is on or after the last row of *T*, then let T_t be a sequence of no rows.

      ii) If the position of *CR* is on a row *R* that is other than the last row of *T*, then let T_t contain all rows of *T* ordered after row *R*, preserving their order in *T*.

iii) If the position of *CR* is before a row *R*, then let T_t contain row *R* and all rows of *T* ordered after row *R*, preserving their order in *T*.

c) If *FO* specifies PRIOR or specifies RELATIVE with a negative value of *J*, then:

i) If *T* is empty or if the position of *CR* is on or before the first row of *T*, then let T_t be a sequence of no rows.

ii) If the position of *CR* is on a row *R* that is other than the first row of *T*, then let T_t contain all rows of *T* ordered before row *R*, preserving their order in *T*.

iii) If the position of *CR* is before a row *R* that is not the first row of *T*, then let T_t contain row all rows of *T* ordered before row *R*, preserving their order in *T*.

iv) If the position of *CR* is after the last row of *T*, then let T_t contain all rows of *T*, preserving their order in *T*.

d) If RELATIVE is specified with a zero value of *J*, then

Case:

i) If the position of *CR* is on a row of *T*, then let T_t be a sequence of rows comprising that one row.

ii) Otherwise, let T_t be an empty sequence of rows.

8) Let *N* be the number of rows in T_t. If *J* is positive, then let *K* be *J*. If *J* is negative, then let *K* be *N*+*J*+1. If *J* is zero and ABSOLUTE is specified, then let *K* be zero; if *J* is zero and RELATIVE is specified, then let *K* be 1 (one).

9) Case:

a) If *K* is greater than 0 (zero) and not greater than *N*, then *CR* is positioned on the *K*-th row of T_t and the corresponding row of *T*. That row becomes the current row of *CR*.

b) Otherwise, a completion condition is raised: *no data*.

Case:

i) If *FO* specifies RELATIVE with *J* equal to 0 (zero), then the position of *CR* is unchanged.

ii) If *FO* implicitly or explicitly specifies NEXT, specifies ABSOLUTE or RELATIVE with *K* greater than *N*, or specifies LAST, then *CR* is positioned after the last row.

iii) Otherwise, *FO* specifies PRIOR, FIRST, or ABSOLUTE, or RELATIVE with *K* not greater than *N* and *CR* is positioned before the first row.

Conformance Rules

None.

15.4 Effect of closing a cursor

Function

Specify the effect of closing a cursor.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* and let *DISP* be the *DISPOSITION* specified in an application of this Subclause.

2) If *CR* is not in the open state, then an exception condition is raised: *invalid cursor state*.

3) Let *RS* be the result set descriptor of *CR*.

4) *CR* is placed in the closed state.

5) Case:

   a) If the operational returnability property of *RS* is WITHOUT RETURN, then *RS* is destroyed.

   b) If *DISP* is DESTROY, then *RS* is removed from the result set sequence that includes *RS*, if any, and *RS* is destroyed.

   NOTE 406 — Otherwise, *RS* is not destroyed.

6) If *CR* is a received cursor, and the result set sequence of the SQL-invoked routine specified in the cursor declaration descriptor of *CR* is not empty, then the General Rules of Subclause 15.2, "Effect of advancing a received cursor to the next result set", are applied.

Conformance Rules

None.

15.5 Effect of a positioned delete

Function

Specify the effect of a positioned delete.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* in an application of this Subclause.

2) If the access mode of the current SQL-transaction or the access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and not every leaf generally underlying table of *CR* is a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction*.

3) If there is any sensitive cursor *SCR*, other than *CR*, that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

   Case:

   a) If *SCR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement is made visible to *SCR* or an exception condition is raised: *cursor sensitivity exception — request failed*.

   b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *SCR* is implementation-defined.

4) If there is any insensitive cursor *ICR*, other than *CR*, that is currently open, then either the change resulting from the successful execution of this statement is invisible to *ICR*, or an exception condition is raised: *cursor sensitivity exception — request failed*.

5) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

6) If *CR* is not positioned on a row, then an exception condition is raised: *invalid cursor state*.

7) If *CR* is a holdable cursor and a <fetch statement> has not been issued against *CR* within the current SQL-transaction, then an exception condition is raised: *invalid cursor state*.

8) Let *T* be the simply underlying table of *CR* and let *LUT* be the leaf underlying table of *T*.

9) Let *R* be the current row of *CR*. Exactly one row *R1* in *LUT* such that each field in *R* is identical to the corresponding field in *R1* is identified for deletion from *LUT*.

NOTE 407 — In case more than one row *R1* satisfies the stated condition, it is implementation-dependent which one is identified for deletion.

NOTE 408 — Identifying a row for deletion is an implementation-dependent mechanism.

10) Whether the current row is removed from the sequence of rows of the result set descriptor of *CR* is implementation-defined.

11) Case:

    a) If *LUT* is a base table, then:

        i) Case:

            1) If <target table> specifies ONLY, then *LUT* is *identified for deletion processing without subtables*.

            2) Otherwise, *LUT* is *identified for deletion processing with subtables*.

            NOTE 409 — Identifying a base table for deletion processing, with or without subtables, is an implementation-dependent mechanism.

        ii) The General Rules of Subclause 15.7, "Effect of deleting rows from base tables", are applied.

    b) If *LUT* is a viewed table, then the General Rules of Subclause 15.9, "Effect of deleting some rows from a viewed table", are applied with <target table> as *VIEW NAME*.

12) If, while *CR* is open, the row from which the current row of *CR* is derived has been marked for deletion by any <delete statement: searched>, <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned>, that identifies any cursor other than *CR*, or updated by any <update statement: searched>, <merge statement>, <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies any cursor other than *CR*, then a completion condition is raised: *warning — cursor operation conflict*.

13) If the <delete statement: positioned> deleted the last row of *CR*, then the position of *CR* is after the last row; otherwise, the position of *CR* is before the next row.

Conformance Rules

None.

15.6 Effect of a positioned update

Function

Specify the effect of a positioned update.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* and let *SCL* be the *SET CLAUSE LIST* in an application of this Subclause.

2) If the access mode of the current SQL-transaction or the access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only and not every leaf generally underlying table of *CR* is a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction*.

3) If there is any sensitive cursor *SCR*, other than *CR*, that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

   Case:

   a) If *SCR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement is made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed*.

   b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *SCR* is implementation-defined.

4) If there is any insensitive cursor *ICR*, other than *CR*, that is currently open, then either the change resulting from the successful execution of this statement is invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed*.

5) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

6) If *CR* is not positioned on a row, then an exception condition is raised: *invalid cursor state*.

7) If *CR* is a holdable cursor and a <fetch statement> has not been issued against *CR* within the current SQL-transaction, then an exception condition is raised: *invalid cursor state*.

8) An object row is any row of a base table from which the current row of *CR* is derived.

9) If, while *CR* is open, an object row has been marked for deletion by any <delete statement: searched>, <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies any cursor other than *CR*, or updated by any <update statement:

searched>, <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies any cursor other than *CR*, or or updated by any <merge statement>, then a completion condition is raised: *warning — cursor operation conflict*.

10) The value associated with DEFAULT is the default value for the <object column> in the containing <set clause> contained in *SCL*, as indicated in the General Rules of Subclause 11.5, "<default clause>".

11) Each <update source> contained in *SCL* is effectively evaluated for the current row before any of the current row's object rows is updated.

12) *CR* remains positioned on its current row, even if an exception condition is raised during evaluation of any <update source>.

13) A candidate new row is constructed by copying the current row of *CR* and updating it as specified by each <set clause> contained in *SCL* by applying the General Rules of Subclause 14.14, "<set clause list>".

NOTE 410 — The data values allowable in an object row may be constrained by a WITH CHECK OPTION constraint. The effect of a WITH CHECK OPTION constraint is defined in the General Rules of Subclause 15.15, "Effect of replacing some rows in a viewed table".

14) Let *CL* be the columns of *T* identified by the <object column>s contained in *SCL*.

15) Let *R1* be the candidate new row and let *R* be the current row of *CR*. Exactly one row *TR* in *T*, such that the value of each field in *R* that is derived from one or more fields in *TR* is identical to the corresponding value that is derived from the same one or more fields in *TR* is identified for replacement in *T*. The current row *R* of *CR* is replaced by *R1*. Let *TR1* be a row consisting of the fields of *R1* and the fields of *TR* that have no corresponding fields in *R1*, ordered according to the order of their corresponding columns in *T*. *TR1* is the replacement row for *TR* and `{ ( TR, TR1 ) }` is the replacement set for *T*.

NOTE 411 — In case more than one row *R1* satisfies the stated condition, it is implementation-dependent which one is identified for replacement.

NOTE 412 — Identifying a row for replacement, associating a replacement row with an identified row, and associating a replacement set with a table are implementation-dependent mechanisms.

16) Case:

a) If *LUT* is a base table, then:

i) Case:

1) If <target table> specifies ONLY, then *LUT* is *identified for replacement processing without subtables* with respect to object columns *CL*.

2) Otherwise, *LUT* is *identified for replacement processing with subtables* with respect to object columns *CL*.

NOTE 413 — Identifying a base table for replacement processing, with or without subtables, is an implementation-dependent mechanism. In general, though not here, the list of object columns can be empty.

ii) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

b) If *LUT* is a viewed table, then the General Rules of Subclause 15.15, "Effect of replacing some rows in a viewed table", are applied with <target table> as *VIEW NAME*.

Conformance Rules

None.

15.7 Effect of deleting rows from base tables

This Subclause is modified by Subclause 15.1, "Effect of deleting rows from base tables", in ISO/IEC 9075-9.

Function

Specify the effect of deleting rows from one or more base tables.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *TT* be the set consisting of every base table that is identified for deletion processing. Let *S* be the set consisting of every row identified for deletion in some table in *TT*.

2) For every row *R* in *S*, every row *SR* that is a subrow or a superrow of *R* is identified for deletion from the base table *BT* containing *SR*, and *BT* is identified for deletion processing.

3) Let *SSC* be the set of state changes in the current trigger execution context.

4) For every table *T* in *TT*, for every table *ST* that is a supertable of *T* or, unless *T* is identified for deletion processing without subtables, a subtable of *T*,

   Case:

   a) If a state change *SC* exists in *SSC* with subject table *ST* and trigger event DELETE, then one copy each of every row of *ST* that is identified for deletion in *ST* is added to the set of transitions of *SC*.

   b) Otherwise, a state change *SC* is added to *SSC* as follows:

      i) The set of transitions of *SC* consists of one copy each of every row of *ST* that is identified for deletion in *ST*.

      ii) The trigger event of *SC* is DELETE.

      iii) The subject table of *SC* is *ST*.

      iv) The column list of *SC* is empty.

      v) The set of statement-level triggers for which *SC* is considered as executed is empty.

      vi) The set of row-level triggers consists of each row-level trigger that is activated by *SC*, paired with the empty set (of rows considered as executed).

5) The Syntax Rules and General Rules of Subclause 15.16, "Execution of BEFORE triggers", are applied with *SSC* as the *SET OF STATE CHANGES*.

6) Every row that is identified for deletion in some table identified for deletion processing is marked for deletion. These rows are no longer identified for deletion, nor are their containing tables identified for deletion processing.

NOTE 414 — “Marking for deletion” is an implementation-dependent mechanism.

Conformance Rules

None.

15.8 Effect of deleting some rows from a derived table

Function

Specify the effect of deleting some rows from a derived table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *QE* be *TABLE* in the application of this Subclause and let *T* be the result of evaluating *QE*.

2) Case:

   a) If *QE* simply contains a <query primary> that immediately contains a <query expression body>, then let *QEB* be that <query expression body>. Apply the General Rules of Subclause 15.8, "Effect of deleting some rows from a derived table", with the table identified by *QEB* as *TABLE*.

   b) If *QE* simply contains a <query expression body> *QEB* that specifies UNION ALL, then let *LO* and *RO* be the <query expression body> and the <query term>, respectively, that are immediately contained in *QEB*. Let *T1* and *T2* be the tables identified by *LO* and *RO*, respectively.

      i) For every row *R* in *T* that has been identified for deletion, let *RD* be the row in either *T1* or *T2* from which *R* has been derived and let *TD* be that table. Identify *RD* for deletion.

      ii) The General Rules of Subclause 15.8, "Effect of deleting some rows from a derived table", are applied with *T1* as *TABLE*.

      iii) The General Rules of Subclause 15.8, "Effect of deleting some rows from a derived table", are applied with *T2* as *TABLE*.

   c) Otherwise, let *QS* be the <query specification> simply contained in *QE*. Let *TE* be the <table expression> immediately contained in *QS*, and *TREF* be the <table reference>s simply contained in the <from clause> of *TE*.

      i) Case:

         1) If *TREF* contains only one <table reference>, then let TR_1 be that <table reference>, and let *m* be 1 (one).

         2) Otherwise, let *m* be the number of <table reference>s that identify tables with respect to which *QS* is one-to-one. Let TR_i, 1 (one) $\leq i \leq m$, be those <table reference>s.

            NOTE 415 — The notion of one-to-one <query specification>s is defined in Subclause 7.12, "<query specification>".

ii) Let TT_i, 1 (one) ≤ i ≤ m, be the table identified by TR_i.

iii) For every row R of T that has been identified for deletion, and for i ranging from 1 (one) to m, let RD be the row in TT_i from which R has been derived. Identify that RD for deletion.

iv) For i ranging from 1 (one) to m,

Case:

1) If TT_i is a base table, then

Case:

A) If TR_i specifies ONLY, then TT_i is identified for deletion processing without subtables.

B) Otherwise, TT_i is identified for deletion processing with subtables.

2) If TT_i is a viewed table, then the General Rules of Subclause 15.9, "Effect of deleting some rows from a viewed table", are applied with TR_i as *VIEW NAME*.

3) Otherwise, the General Rules of Subclause 15.8, "Effect of deleting some rows from a derived table", are applied with TR_i as *TABLE*.

v) The General Rules of Subclause 15.7, "Effect of deleting rows from base tables", are applied.

Conformance Rules

None.

15.9 Effect of deleting some rows from a viewed table

Function

Specify the effect of deleting some rows from a viewed table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *VN* be the *VIEW NAME* in an application of this Subclause.

2) If *VN* specifies ONLY, then let *QE* be the original <query expression> included in the descriptor of the view *V* identified by *VN*; otherwise, let *QE* be the <query expression> contained in that descriptor. Let *T* be the result of evaluating *QE*.

3) For each row *R* of *V* that has been identified for deletion, let *RD* be the row in *T* from which *R* has been derived; identify that row for deletion.

4) The General Rules of Subclause 15.8, "Effect of deleting some rows from a derived table", are applied with *QE* as *TABLE*.

Conformance Rules

None.

15.10 Effect of inserting tables into base tables

This Subclause is modified by Subclause 15.2, "Effect of inserting tables into base tables", in ISO/IEC 9075-9.

Function

Specify the effect of inserting each of one or more given tables into its associated base table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *SSC* be the set of state changes in the current trigger execution context.

2) For each base table *T* that is identified for insertion, let *S* be the source table for *T*.

   a) If some column *IC* of *T* is the identity column of *T*, then for each row in *S* whose site *ICS* corresponding to *IC* is marked as *unassigned*:

      i) *ICS* is no longer marked as unassigned.

      ii) Let *NV* be the result of applying the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator", with the sequence descriptor included in the column descriptor of *IC* as *SEQUENCE*.

         Case:

         1) If the declared type of *IC* is a distinct type *DIST*, then let *ICNV* be *DIST*(*NV*).

         2) Otherwise, let *ICNV* be *NV*.

      iii) The General Rules of Subclause 9.2, "Store assignment", are applied with *ICS* as *TARGET* and *ICNV* as *VALUE*.

   b) Every proper supertable *ST* of *T* is identified for insertion. A source table for insertion into each *ST* is constructed as follows:

      i) Let *S* be the source table for the insertion into *T*. Let *TVC* be some <table value constructor> whose value is *S*.

      ii) Let *n* be the number of column descriptors included in the table descriptor of *ST* and let CD_i, 1 (one) $\leq i \leq n$, be those column descriptors. Let *SL* be a <select list> containing *n* <select sublist>s such that, for *i* ranging from 1 (one) to *n*, the *i*-th <select sublist> consists of the column name included in CD_i.

iii) The source table for insertion into *ST* consists of the rows in the result of the <query expression>:

```
SELECT SL FROM TVC
```

3) For every base table *BT* that is identified for insertion,

Case:

a) If a state change *SC* exists in *SSC* with subject table *ST* and trigger event INSERT, then the rows in the source table for *BT* are added to the set of transitions of *SC*.

b) Otherwise, a state change *SC* is added to *SSC* as follows:

i) The set of transitions of *SC* consists of the rows in the source table for *BT*.

ii) The trigger event of *SC* is INSERT.

iii) The subject table of *SC* is *BT*.

iv) The column list of *SC* is empty.

v) The set of statement-level triggers for which *SC* is considered as executed is empty.

vi) The set of row-level triggers consists of each row-level trigger that is activated by *SC*, paired with the empty set (of rows considered as executed).

4) The Syntax Rules and General Rules of Subclause 15.16, "Execution of BEFORE triggers", are applied with *SSC* as the *SET OF STATE CHANGES*.

5) For every state change *SC* in *SSC*, let *SOT* be the set of transitions in *SC* and let *BT* be the subject table of *SC*.

a) In each row *R* in *SOT*, for each site *GCS* in *R* corresponding to a generated column *GC*, let *GCR* be the result of evaluating, for R, the generation expression included in the column descriptor of *GC*. The General Rules of Subclause 9.2, "Store assignment", are applied with *GCS* as *TARGET* and *GCR* as *VALUE*.

b) Every row in *SOT* is inserted into *BT* and *BT* is no longer identified for insertion.

Conformance Rules

None.

15.11 Effect of inserting a table into a derived table

Function

Specify the effect of inserting a table into a derived table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *Q* and *T* be the *SOURCE* and *TARGET*, respectively, in the application of this Subclause.

2) Let *QE* be the <query expression> included in the descriptor of *T*.

   Case:

   a) If *QE* simply contains a <query primary> that immediately contains a <query expression body>, then let *QEB* be that <query expression body>. Apply the General Rules of Subclause 15.11, "Effect of inserting a table into a derived table", with *Q* as *SOURCE* and the result of *QEB* as *TARGET*.

   b) Otherwise, let *QS* be the <query specification> simply contained in *QE*. Let *TE* be the <table expression> immediately contained in *QS*, and *TREF* be the <table reference>s simply contained in the <from clause> of *TE*. Let *SL* be the <select list> immediately contained in *QS*, and *n* the number of <value expression>s VE_j, 1 (one) $\leq j \leq n$, simply contained in *SL*.

      i) Case:

         1) If *TREF* contains only one <table reference>, then let TR_1 be that <table reference>, and let *m* be 1 (one).

         2) Otherwise, let *m* be the number of <table reference>s that identify tables with respect to which *QS* is one-to-one. Let TR_i, 1 (one) $\leq i \leq m$, be those <table reference>s.

      ii) Let TT_i, 1 (one) $\leq i \leq m$, be the table identified by TR_i, and let S_i be an initially empty table of candidate rows for TT_i.

      iii) For every row *R* of *Q*, and for *i* ranging from 1 (one) to *m*:

         1) A candidate row of TT_i is effectively created in which the value of each column is its default value, as specified the General Rules of Subclause 11.5, "<default clause>". The candidate row includes every column of TT_i.

2) For j ranging from 1 (one) to n, let C be a column of some candidate row identified by VE_j, and let SV be the j-th value of R. The General Rules of Subclause 9.2, "Store assignment", are applied to C and SV as *TARGET* and *SOURCE*, respectively.

3) The candidate row is added to the corresponding S_i.

iv) For i ranging from 1 (one) to m,

Case:

1) If TT_i is a base table, then TT_i is identified for insertion of source table S_i.

2) If TT_i is a viewed table, the General Rules of Subclause 15.12, "Effect of inserting a table into a viewed table", are applied with S_i as *SOURCE* and TT_i as *TARGET*.

3) Otherwise, the General Rules of Subclause 15.11, "Effect of inserting a table into a derived table", are applied with S_i as *SOURCE* and TT_i as *TARGET*.

v) The General Rules of Subclause 15.10, "Effect of inserting tables into base tables", are applied.

Conformance Rules

None.

15.12 Effect of inserting a table into a viewed table

Function

Specify the effect of inserting a table into a viewed table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* and *T* be the *SOURCE* and *TARGET*, respectively, in application of this Subclause. Let *TD* be the view descriptor of *T*. Let *QE* be the original <query expression> included in *TD*.

2) Case:

   a) If *TD* indicates WITH CHECK OPTION, then:

      i) Case:

         1) If *TD* specifies LOCAL, then let *VD* be a view descriptor derived from *TD* by removing the WITH CHECK OPTION indication.

         2) Otherwise, let *VD* be a view descriptor derived from *TD* as follows:

            A) The WITH CHECK OPTION indication is removed.

            B) Every reference contained in *QE* to an underlying table *UV* of *QE* that is a viewed table is replaced by a reference to a view whose descriptor is identical to that of *UV* except that WITH CASCADED CHECK OPTION is indicated.

      ii) The General Rules of this Subclause are applied with *S* as *SOURCE* and the view described by *VD* as *TARGET*.

      iii) If the result of

```
EXISTS ( SELECT * FROM S
         EXCEPT ALL
         SELECT * FROM T )
```

      is *True*, then an exception condition is raised: *with check option violation*.

   b) Otherwise, the General Rules of Subclause 15.11, "Effect of inserting a table into a derived table", are applied, with *S* as *SOURCE* and *QE* as *TARGET*.

Conformance Rules

None.

15.13 Effect of replacing rows in base tables

This Subclause is modified by Subclause 15.3, "Effect of replacing rows in base tables", in ISO/IEC 9075-9.

Function

Specify the effect of replacing some of the rows in one or more base tables.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *TT* be the set consisting of every base table that is identified for replacement processing. Let *S* be the set consisting of every row identified for replacement in every table in *TT*.

2) For every base table *T* in *TT*, let *OC* be the set consisting of every object column with respect to which *T* is identified for replacement processing and every generated column of *T* that depends on at least one of these object columns. Every table *ST* that is a subtable or supertable of *T* is identified for replacement processing with respect to the intersection (possibly empty) of *OC* and the columns of *ST*.

3) For every row *R* that is identified for replacement in some table *T* in *TT*, every row *SR* that is a subrow or a superrow of *R* is identified for replacement in the base table *ST* that contains *SR*. The replacement set *RST* for *ST* is derived from the replacement set *RR* for *T* as follows.

   Case:

   a) If *ST* is a subtable of *T*, then each replacement row in *RST* is the corresponding replacement row in *RR* extended with those fields of the corresponding identified row in *ST* that have no corresponding column in *T*.

   b) If *ST* is a supertable of *T*, then each replacement row in *RST* is the corresponding replacement row in *RR* minus those fields that have no corresponding column in *ST*.

4) Let *SSC* be the set of state changes in the current trigger execution context.

5) For every table *ST* that is identified for replacement processing, let *TL* be the set consisting of the names of the columns of *ST*. For every subset *STL* of *TL* such that either *STL* is empty or the intersection of *STL* and *OC* is not empty:

   a) If some column *IC* of *T* is the identity column of *ST*, then, for each row identified for replacement in *ST* whose site *ICS* corresponding to *IC* is marked as *unassigned*:

i) Let *NV* be the result of applying the General Rules of Subclause 9.23, "Generation of the next value of a sequence generator", with the sequence descriptor included in the column descriptor of *IC* as *SEQUENCE*.

Case:

1) If the declared type of *IC* is a distinct type *DIST*, then let *ICNV* be *DIST*(*NV*).

2) Otherwise, let *ICNV* be *NV*.

ii) The General Rules of Subclause 9.2, "Store assignment", are applied with *ICS* as *TARGET* and *ICNV* as *VALUE*.

b) All sites in *ST* that are marked as unassigned cease to be so marked.

c) Case:

i) If a state change *SC* exists in *SSC* with subject table *ST*, trigger event UPDATE, and column list *STL*, then the row pairs formed by pairing each row identified for replacement in *ST* with its corresponding replacement row are added to the set of transitions of *SC*.

ii) Otherwise, a state change *SC* is added to *SSC* as follows:

1) The set of transitions of *SC* consists of row pairs formed by pairing each row identified for replacement in *ST* with its corresponding replacement row.

2) The trigger event of *SC* is UPDATE.

3) The subject table of *SC* is *ST*.

4) The column list of *SC* is *STL*.

5) The set of statement-level triggers for which *SC* is considered as executed is empty.

6) The set of row-level triggers consists of each row-level trigger that is activated by *SC*, paired with the empty set (of rows considered as executed).

6) The Syntax Rules and General Rules of Subclause 15.16, "Execution of BEFORE triggers", are applied with *SSC* as the *SET OF STATE CHANGES*.

7) For each set of transitions *RST* in each state change *SC* in *SSC*, in each row *R* in *RST*, for each site *GCS* in *R* corresponding to a generated column *GC* in the subject table of *SC*, let *GCR* be the result of evaluating, for *R*, the generation expression included in the column descriptor of *GC*. The General Rules of Subclause 9.2, "Store assignment", are applied with *GCS* as *TARGET* and *GCR* as *VALUE*.

8) For every table *T* in *TT*, for every table *ST* that is a supertable or a subtable of *T*, for every row *R* that is identified for replacement in *ST*, *R* is replaced by its new transition variable. *R* is no longer identified for replacement. *ST* is no longer identified for replacement processing.

Conformance Rules

None.

15.14 Effect of replacing some rows in a derived table

Function

Specify the effect of replacing some rows in a derived table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *QE* be the *TABLE* and *RS* the replacement for *TABLE* in the application of this Subclause.

2) Let *T* be the result of evaluating *QE*. Let *CL* be the object columns of *QE*.

3) Case:

   a) If *QE* simply contains a <query primary> that immediately contains a <query expression body>, then let *QEB* be that <query expression body>. Apply the General Rules of Subclause 15.14, "Effect of replacing some rows in a derived table", with *TR* as the table identified by *QEB*, and with *RS* as the replacement set for *TR*.

   b) If *QE* simply contains a <query expression body> *QEB* that specifies UNION ALL, let *LO* and *RO* be the <query expression body> and the <query term>, respectively, that are immediately contained in *QEB*. Let *T1* and *T2* be the tables identified by *LO* and *RO*, respectively. Let the columns of *T1* and *T2* that are underlying columns of the object columns of *CL* be the object columns *CL1* and *CL2*, respectively. Let *RS1* and *RS2* be the initially empty replacement sets for *T1* and *T2*, respectively.

      i) For every pair (*SR*, *CNR*) of *RS*,

         Case:

         1) If *SR* has been derived from a row of *T1*, then identify that row *SR1* for replacement by *CNR*; the pair (*SR1*, *CNR*) is effectively added to *RS1*.

         2) Otherwise, let *SR2* be the row of *T2* from which *SR* has been derived; identify that row for replacement by *CNR*; the pair (*SR2*, *CNR*) is effectively added to *RS2*.

      ii) The General Rules of Subclause 15.14, "Effect of replacing some rows in a derived table", are applied with *T1* as *TABLE*.

      iii) The General rules of Subclause 15.14, "Effect of replacing some rows in a derived table", are applied with *T2* as *TABLE*.

   c) Otherwise, let *QS* be the <query specification> simply contained in *QE*. Let *TE* be the <table expression> immediately contained in *QS*, and let *TREF* be the <table reference>s simply contained in the <from

clause> of *TE*. Let *SL* be the <select list> immediately contained in *QS*, and let *n* be the number of <value expression>s VE_j, 1 (one) ≤ *j* ≤ *n*, simply contained in *SL*.

i) Case:

1) If *TREF* contains only one <table reference>, then let TR_1 be that <table reference>, and let *m* be 1 (one).

2) Otherwise, let *m* be the number of <table reference>s that identify tables with respect to which *QS* is one-to-one. Let TR_i, 1 (one) ≤ *i* ≤ *m*, be those <table reference>s.

ii) Let TT_i, 1 (one) ≤ *i* ≤ *m*, be the table identified by TR_i, let RS_i be an initially empty replacement set for TT_i, and let CL_i be the object column list of TT_i, such that every column of CL_i is an underlying column of *CL*.

iii) For every pair (*SR*, *CNR*) of *RS*, and for *i* ranging from 1 (one) to *m*:

1) Let *SRTI* be the row of TT_i from which *SR* has been derived.

2) A candidate row *CNRI* of TT_i is effectively created in which the value of each column is its default value, as specified the General Rules of Subclause 11.5, "<default clause>". The candidate row includes every column of TT_i.

3) For *j* ranging from 1 (one) to *n*, let *C* be a column of some candidate row identified by VE_j, and let *SV* be the *j*-th value of *CNR*. The General Rules of Subclause 9.2, "Store assignment", are applied to *C* and *SV* as *TARGET* and *SOURCE*, respectively.

4) Identify *SRTI* for replacement by *CNRI*; the pair (*SRTI*, *CNRI*) is effectively added to SR_i.

iv) For *i* ranging from 1 (one) to *m*

Case:

1) If TT_i is a base table, then

Case:

A) If TR_i specifies ONLY, then TT_i is identified for replacement processing without subtables with respect to the object columns CL_i.

B) Otherwise, TT_i is identified for replacement processing with subtables with respect to the object columns CL_i.

2) If TT_i is a viewed table, then the General rules of Subclause 15.15, "Effect of replacing some rows in a viewed table", are applied with TR_i as *VIEW NAME*.

3) If TT_i is a derived table, then the General rules of Subclause 15.14, "Effect of replacing some rows in a derived table", are applied with TR_i as *TABLE*.

v) The General Rules of Subclause 15.13, "Effect of replacing rows in base tables", are applied.

Conformance Rules

None.

15.15 Effect of replacing some rows in a viewed table

Function

Specify the effect of replacing some rows in a viewed table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *T* be the *VIEW NAME* and *RS* the replacement set for *VIEW NAME* in application of this Subclause. Let *TD* be the view descriptor of *T*. If *VN* specifies ONLY, then let *QE* be the original <query expression> included in *TD*; otherwise, let *QE* be the <query expression> included in *TD*.

2) Case:

   a) If *TD* indicates WITH CHECK OPTION, then:

      i) Case:

         1) If *TD* specifies LOCAL, then let *VD* be a view descriptor derived from *TD* by removing the WITH CHECK OPTION indication.

         2) Otherwise, let *VD* be a view descriptor derived from *TD* as follows:

            A) The WITH CHECK OPTION indication is removed.

            B) Every reference contained in *QE* to an underlying table *UV* of *QE* that is a viewed table is replaced by a reference to a view whose descriptor is identical to that of *UV* except that WITH CASCADED CHECK OPTION is indicated.

      ii) The General Rules of this Subclause are applied with the view *V* described by *VD* as *VIEW NAME* and *RS* as the replacement set for *V*.

      iii) Let *S* be the table consisting of the candidate new rows of *RS*. If the result of

```
EXISTS ( SELECT * FROM S
         EXCEPT ALL
         SELECT * FROM T )
```

      is <u>*True*</u>, then an exception condition is raised: *with check option violation*.

   b) Otherwise, the General Rules of Subclause 15.14, "Effect of replacing some rows in a derived table", are applied with *QE* as *TABLE* and *RS* as the replacement set for *QE*.

Conformance Rules

None.

15.16 Execution of BEFORE triggers

Function

Define the execution of BEFORE triggers.

Syntax Rules

1) Let *SSC* be the *SET OF STATE CHANGES* specified in an application of this Subclause.

2) Let *BT* be the set of BEFORE triggers that are activated by some state change in *SSC*.

   NOTE 416 — Activation of triggers is defined in Subclause 4.38, "Triggers".

3) Let *NT* be the number of triggers in *BT* and let TR_k be the *k*-th such trigger, ordered according to their order of execution. Let SC_k be the state change in *SSC* that activated TR_k.

   NOTE 417 — Ordering of triggers is defined in Subclause 4.38, "Triggers".

Access Rules

None.

General Rules

1) For *k* ranging from 1 (one) to *NT*, apply the General Rules of Subclause 15.18, "Execution of triggers", with TR_k as *TRIGGER* and SC_k as *STATE CHANGE*, respectively.

Conformance Rules

None.

15.17 Execution of AFTER triggers

Function

Define the execution of AFTER triggers.

Syntax Rules

1) Let *SSC* be the *SET OF STATE CHANGES* specified in an application of this Subclause.

2) Let *AT* be the set of AFTER triggers that are activated by some state change in *SSC*.

   NOTE 418 — Activation of triggers is defined in Subclause 4.38, "Triggers".

3) Let *NT* be the number of triggers in *AT* and let TR_k be the *k*-th such trigger, ordered according to their order of execution. Let SC_k be the state change in *SSC* that activated TR_k.

   NOTE 419 — Ordering of triggers is defined in Subclause 4.38, "Triggers".

Access Rules

None.

General Rules

1) For *k* ranging from 1 (one) to *NT*, apply the General Rules of Subclause 15.18, "Execution of triggers", with TR_k as *TRIGGER* and SC_k as *STATE CHANGE*, respectively.

Conformance Rules

None.

15.18 Execution of triggers

Function

Define the execution of triggers.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *TR* and *SC* be respectively a *TRIGGER* and a *STATE CHANGE* in an application of this Subclause.

2) Let *TA* be the triggered action included in the trigger descriptor of *TR*. Let *TSS* be the <triggered SQL statement> contained in *TA*. Let *TE* be the trigger event of *SC*. Let *ST* be the set of transitions in *SC*.

3) *TR* is executed as follows:

   Case:

   a) If *TR* is a row-level trigger, then, for each transition *T* in *ST* for which *TR* is not considered as executed, *TA* is invoked and *TR* is considered as executed for *T*. The order in which the transitions in *ST* are taken is implementation-dependent.

   b) If *TR* is not considered as executed for *SC*, then *TA* is invoked once and *TR* is considered as executed for *SC*.

4) When *TA* is invoked:

   a) Case:

      i) If *TE* is DELETE, then the old transition table for the invocation of *TA* is *ST*. If *TR* is a row-level trigger, then the value of the old transition variable for the execution of *TSS* is *T*.

      ii) If *TE* is INSERT, then the new transition table for the invocation of *TA* is *ST*. If *TR* is a row-level trigger, then the value of the new transition variable for the invocation of *TA* is *T*.

      iii) If *TE* is UPDATE, then the old transition table for the invocation of *TA* is the multiset formed by taking the old rows of the transitions in in *ST* and the new transition table for the invocation of *TA* is the multiset formed by taking the new rows of the transitions in *ST*. If *TR* is a row-level trigger, then the value of the old transition variable for the invocation of *TA* is the old row of *T* and the new transition variable for the invocation of *TA* is the new row of *T*.

   b) Case:

      i) If *TA* contains a <search condition> *TASC* and the result of evaluating *TASC* is *True*, then *TSS* is executed.

ii) If *TA* does not contain a <search condition>, then *TSS* is executed.

5) When *TSS* is executed:

a) The General Rules of Subclause 23.2, "Pushing and popping the diagnostics area stack", are applied with "PUSH" as *OPERATION* and the diagnostics area stack as *STACK*.

b) The authorization identifier of the owner of the schema that includes the trigger descriptor of *TR* is pushed onto the authorization stack.

c) A new savepoint level is established.

d) Let *N* be the number of <SQL procedure statement>s simply contained in *TSS*. For *i* ranging from 1 (one) to *N*:

i) Let S_i be the *i*-th such <SQL procedure statement>.

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

e) The <SQL procedure statement>s simply contained in *TSS* are effectively executed in the order in which they are specified in *TSS*.

f) If, before the completion of the execution of any <SQL procedure statement> simply contained in *TSS*, an attempt is made to execute an SQL-schema statement, an SQL-dynamic statement, or an SQL-session statement then an exception condition is raised: *prohibited statement encountered during trigger execution*.

g) If *TR* is a BEFORE trigger and if, before the completion of the execution of any <SQL procedure statement> simply contained in *TSS*, an attempt is made to execute an SQL-data change statement or an SQL-invoked routine that possibly modifies SQL-data, then an exception condition is raised: *prohibited statement encountered during trigger execution*.

h) The current savepoint level is destroyed.

NOTE 420 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

i) The General Rules of Subclause 23.2, "Pushing and popping the diagnostics area stack", are applied with "POP" as *OPERATION* and the diagnostics area stack as *STACK*.

j) The top cell in the authorization stack is removed.

k) If the execution of *TSS* is not successful, then an exception condition is raised: *triggered action exception*. The exception condition that caused *TSS* to fail is raised.

NOTE 421 — Raising the exception condition that caused *TSS* to fail enters the exception information into the diagnostics area that was pushed prior to the execution of *TSS*.

Conformance Rules

None.

(Blank page)

16 Control statements

This Clause is modified by Clause 14, "Control statements", in ISO/IEC 9075-4.
This Clause is modified by Clause 15, "Control statements", in ISO/IEC 9075-14.

16.1 <call statement>

This Subclause is modified by Subclause 11.22, "<call statement>", in ISO/IEC 9075-10.

Function

Invoke an SQL-invoked routine.

Format

```
<call statement> ::=
  CALL <routine invocation>
```

Syntax Rules

1) Let *RI* be the <routine invocation> immediately contained in the <call statement>.

2) The Syntax Rules of Subclause 10.4, "<routine invocation>", are invoked with *RI* as *ROUTINE INVOCATION*, yielding subject routine *SR* and static SQL argument list *SAL*.

3) *SR* shall be an SQL-invoked procedure.

Access Rules

None.

General Rules

1) The General Rules of Subclause 10.4, "<routine invocation>", are invoked with *SR* as *ROUTINE INVOCATION* and *SAL* as *STATIC SQL ARG LIST*.

Conformance Rules

None.

16.2 <return statement>

This Subclause is modified by , in ISO/IEC 9075-14.

Function

Return a value from an SQL function.

Format

```
<return statement> ::=
  RETURN <return value>

<return value> ::=
    <value expression>
  | NULL
```

Syntax Rules

1) <return statement> shall be contained in an SQL routine body that is simply contained in the <routine body> of an <SQL-invoked function> *F*. Let *RDT* be the <returns data type> of the <returns clause> of *F*.

2) The <return value> <null specification> is equivalent to the <value expression>:

   `CAST (NULL AS RDT)`

3) Let *VE* be the <value expression> of the <return value> immediately contained in <return statement>.

4) The declared type of *VE* shall be assignable to an item of the data type *RDT*, according to the Syntax Rules of Subclause 9.2, "Store assignment", with *RDT* and *VE* as *TARGET* and *VALUE*, respectively.

Access Rules

None.

General Rules

1) The value of *VE* is the *returned value* of the execution of the SQL routine body of *F*.

2) The execution of the SQL routine body of *F* is terminated immediately.

Conformance Rules

None.

17 Transaction management

17.1 <start transaction statement>

Function

Start an SQL-transaction and set its characteristics.

Format

```
<start transaction statement> ::=
  START TRANSACTION [ <transaction characteristics> ]
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) If an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction*.

2) Case:

   a) If <transaction characteristics> is omitted, then let *TC* be

      ECAM ISOLATION LEVEL *ECIL* DIAGNOSTICS SIZE *ECNC*

      where *ECAM*, *ECIL*, and *ECNC* are the access mode, isolation level and number of conditions, respectively, of the enduring transaction characteristics of the current SQL-session.

   b) Otherwise, let *TC* be the <transaction characteristics>.

3) If <number of conditions> is specified and is less than 1 (one), then an exception condition is raised: *invalid condition number*.

4) The <set transaction statement>

   SET TRANSACTION *TC*

is effectively executed.

NOTE 422 — The characteristics of a transaction begun by a <start transaction statement> are as specified here regardless of the characteristics specified by any preceding <set transaction statement>. That is, even if one or more characteristics are omitted by the <start transaction statement>, the defaults specified in the Syntax Rules of this Subclause and of Subclause 17.3, "<transaction characteristics>", are effective and are not affected by any (preceding) <set transaction statement> in the same SQL-session.

5) An SQL-transaction is initiated.

Conformance Rules

1) Without Feature T241, "START TRANSACTION statement", conforming SQL language shall not contain a <start transaction statement>.

17.2 <set transaction statement>

This Subclause is modified by Subclause 11.21, "<set transaction statement>", in ISO/IEC 9075-10.

Function

Set the characteristics of the next SQL-transaction for the SQL-agent.

NOTE 423 — This statement has no effect on any SQL-transactions subsequent to the next SQL-transaction.

Format

```
<set transaction statement> ::=
  SET [ LOCAL ] TRANSACTION <transaction characteristics>
```

Syntax Rules

1) If LOCAL is specified, then <transaction characteristics> shall not contain <number of conditions>.

Access Rules

None.

General Rules

1) Case:

   a) If a <set transaction statement> that does not specify LOCAL is executed, then

      Case:

      i) If an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction*.

      ii) If an SQL-transaction is not currently active, then if there are any holdable cursors remaining open from the previous SQL-transaction and the isolation level of the previous SQL-transaction is not the same as the isolation level determined by the <level of isolation>, then an exception condition is raised: *invalid transaction state — held cursor requires same isolation level*.

   b) If a <set transaction statement> that specifies LOCAL is executed, then:

      i) If the SQL-implementation does not support SQL-transactions that affect more than one SQL-server, then an exception condition is raised: *feature not supported — multiple server transactions*.

      ii) If there is no SQL-transaction that is currently active, then an exception condition is raised: *invalid transaction state — no active SQL-transaction for branch transaction*.

      iii) If there is an active SQL-transaction and there has been a transaction-initiating SQL-statement executed at the current SQL-connection in the context of the active SQL-transaction, then an exception condition is raised: *invalid transaction state — branch transaction already active*.

iv) If the transaction access mode of the SQL-transaction is read-only and <transaction access mode> specifies READ WRITE, then an exception condition is raised: *invalid transaction state — inappropriate access mode for branch transaction*.

v) If the isolation level of the SQL-transaction is SERIALIZABLE and <level of isolation> specifies anything except SERIALIZABLE, then an exception condition is raised: *invalid transaction state — inappropriate isolation level for branch transaction*.

vi) If the isolation level of the SQL-transaction is REPEATABLE READ and <level of isolation> specifies anything except REPEATABLE READ or SERIALIZABLE, then an exception condition is raised: *invalid transaction state — inappropriate isolation level for branch transaction*.

vii) If the isolation level of the SQL-transaction is READ COMMITTED and <level of isolation> specifies READ UNCOMMITTED, then an exception condition is raised: *invalid transaction state — inappropriate isolation level for branch transaction*.

NOTE 424 — If the isolation level of the SQL-transaction is READ UNCOMMITTED, then any <level of isolation> is permissible.

2) If <number of conditions> is specified and is less than 1 (one), then an exception condition is raised: *invalid condition number*.

3) Let *TC* be <transaction characteristics>. Let *CSC* be the current SQL-session context.

4) If the explicit or implicit <transaction access mode> contains READ ONLY, then the current access mode of *CSC* is set to *read-only*. Otherwise, the current access mode of *CSC* is set to *read-write*.

5) The current isolation level of *CSC* is set to an implementation-defined isolation level that will not exhibit any of the phenomena that the explicit or implicit <level of isolation> contained in *TC* would not exhibit, as specified in Table 8, “SQL-transaction isolation levels and the three phenomena”.

6) The current condition area limit of *CSC* is set to the explicit or implicit <number of conditions> contained in *TC*.

Conformance Rules

1) Without Feature T251, “SET TRANSACTION statement: LOCAL option”, conforming SQL language shall not contain a <set transaction statement> that immediately contains LOCAL.

17.3 <transaction characteristics>

Function

Specify transaction characteristics.

Format

```
<transaction characteristics> ::=
      [ <transaction mode> [ { <comma> <transaction mode> }... ] ]

<transaction mode> ::=
    <isolation level>
  | <transaction access mode>
  | <diagnostics size>

<transaction access mode> ::=
    READ ONLY
  | READ WRITE

<isolation level> ::=
  ISOLATION LEVEL <level of isolation>

<level of isolation> ::=
    READ UNCOMMITTED
  | READ COMMITTED
  | REPEATABLE READ
  | SERIALIZABLE

<diagnostics size> ::=
  DIAGNOSTICS SIZE <number of conditions>

<number of conditions> ::=
  <simple value specification>
```

Syntax Rules

1) Let TC be the <transaction characteristics>.

2) TC shall contain at most one <isolation level>, at most one <transaction access mode>, and at most one <diagnostics size>.

3) If TC does not contain an <isolation level>, then ISOLATION LEVEL SERIALIZABLE is implicit.

4) If <transaction access mode> is READ WRITE, then the <level of isolation> shall not be READ UNCOMMITTED.

5) If TC does not contain a <transaction access mode>, then

   Case:

   a) If <isolation level> contains READ UNCOMMITTED, then READ ONLY is implicit.

   b) Otherwise, READ WRITE is implicit.

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

7) If *TC* does not contain a <diagnostics size>, then DIAGNOSTICS SIZE *n* is implicit, where *n* is an implementation-dependent value not less than 1 (one).

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F111, “Isolation levels other than SERIALIZABLE”, conforming SQL language shall not contain an <isolation level> that contains a <level of isolation> other than SERIALIZABLE.

2) Without Feature F121, “Basic diagnostics management”, conforming SQL language shall not contain a <diagnostics size>.

17.4 <set constraints mode statement>

Function

If an SQL-transaction is currently active, then set the constraint mode for that SQL-transaction in the current SQL-session. If no SQL-transaction is currently active, then set the constraint mode for the next SQL-transaction in the current SQL-session for the SQL-agent.

NOTE 425 — This statement has no effect on any SQL-transactions subsequent to this SQL-transaction.

Format

```
<set constraints mode statement> ::=
  SET CONSTRAINTS <constraint name list> { DEFERRED | IMMEDIATE }

<constraint name list> ::=
    ALL
  | <constraint name> [ { <comma> <constraint name> }... ]
```

Syntax Rules

1) If a <constraint name> is specified, then it shall identify a constraint.

2) The constraint identified by <constraint name> shall be deferrable.

Access Rules

None.

General Rules

1) Let *CSC* be the current SQL-session context.

2) If IMMEDIATE is specified, then

   Case:

   a) If ALL is specified, then the constraint mode in *CSC* of all constraints that are deferrable is set to immediate.

   b) Otherwise, the constraint mode in *CSC* for the constraints identified by the <constraint name>s in the <constraint name list> is set to immediate.

3) If DEFERRED is specified, then

   Case:

   a) If ALL is specified, then the constraint mode in *CSC* of all constraints that are deferrable is set to deferred.

   b) Otherwise, the constraint mode in *CSC* for the constraints identified by the <constraint name>s in the <constraint name list> is set to deferred.

Conformance Rules

1) Without Feature F721, “Deferrable constraints”, conforming SQL language shall not contain a <set constraints mode statement>.

17.5 <savepoint statement>

This Subclause is modified by Subclause 11.17, "<savepoint statement>", in ISO/IEC 9075-10.

Function

Establish a savepoint.

Format

```
<savepoint statement> ::=
  SAVEPOINT <savepoint specifier>

<savepoint specifier> ::=
  <savepoint name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the <savepoint name>.

2) If *S* identifies an existing savepoint established within the current savepoint level, then that savepoint is destroyed.

3) If the number of savepoints that now exist within the current SQL-transaction is equal to the implementation-defined maximum number of savepoints per SQL-transaction, then an exception condition is raised: *savepoint exception — too many*.

4) A savepoint is established in the current savepoint level and at the current point in the current SQL-transaction. *S* is assigned as the identifier of that savepoint.

Conformance Rules

1) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <savepoint statement>.

17.6 <release savepoint statement>

This Subclause is modified by Subclause 11.18, "<release savepoint statement>", in ISO/IEC 9075-10.

Function

Destroy a savepoint.

Format

```
<release savepoint statement> ::=
  RELEASE SAVEPOINT <savepoint specifier>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the <savepoint name>.

2) If *S* does not identify a savepoint established in the current savepoint level, then an exception condition is raised: *savepoint exception — invalid specification*.

3) The savepoint identified by *S* and all savepoints established in the current savepoint level subsequent to the establishment of *S* are destroyed.

Conformance Rules

1) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <release savepoint statement>.

17.7 <commit statement>

This Subclause is modified by Subclause 11.19, "<commit statement>", in ISO/IEC 9075-10.

Function

Terminate the current SQL-transaction with commit.

Format

```
<commit statement> ::=
  COMMIT [ WORK ] [ AND [ NO ] CHAIN ]
```

Syntax Rules

1) If neither AND CHAIN nor AND NO CHAIN is specified, then AND NO CHAIN is implicit.

Access Rules

None.

General Rules

1) If the current SQL-transaction is part of an encompassing transaction that is controlled by an agent other than the SQL-agent, then an exception condition is raised: *invalid transaction termination*.

2) If an atomic execution context is active, then an exception condition is raised: *invalid transaction termination*.

3) For every open cursor *CR* that is not a holdable cursor in the stack of contexts of the current SQL-session, the General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

4) For every temporary table in any SQL-client module associated with the current SQL-transaction that specifies the ON COMMIT DELETE option and that was updated by the current SQL-transaction, the execution of the <commit statement> is effectively preceded by the execution of a <delete statement: searched> that specifies DELETE FROM *T*, where *T* is the <table name> of that temporary table.

5) The effects specified in the General Rules of Subclause 17.4, "<set constraints mode statement>" occur as if the statement SET CONSTRAINTS ALL IMMEDIATE were executed for each active SQL-connection.

6) Case:

   a) If any constraint is not satisfied, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback — integrity constraint violation*.

   b) If the execution of any <triggered SQL statement> is unsuccessful, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback — triggered action exception*.

c) If any other error preventing commitment of the SQL-transaction has occurred, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback* with an implementation-defined subclass value.

d) Otherwise, any changes to SQL-data or schemas that were made by the current SQL-transaction are eligible to be perceived by all concurrent and subsequent SQL-transactions.

7) All savepoint levels are destroyed and a new savepoint level is established.

NOTE 426 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

8) Every valid non-holdable locator value is marked invalid.

9) The current SQL-transaction is terminated.

10) Case:

a) If <commit statement> contains AND CHAIN, then an SQL-transaction is initiated. Any branch transactions of the SQL-transaction are initiated with the same access mode, isolation level, and diagnostics area limit as the corresponding branch of the SQL-transaction just terminated.

b) Otherwise:

i) The current access mode, current isolation level, and current condition area limit of the current SQL-session context are set to the access mode, isolation level, and condition area limit, respectively, of the enduring transaction characteristics of the current SQL-session.

ii) For every constraint *C*, the constraint mode of *C* in the current SQL-session context is set to the initial constraint mode included in the constraint descriptor for *C*.

11) The prepared statement of every held cursor remains in existence. It is implementation-defined whether or not any other prepared statement is deallocated.

Conformance Rules

1) Without Feature T261, “Chained transactions”, conforming SQL language shall not contain a <commit statement> that immediately contains CHAIN.

17.8 <rollback statement>

This Subclause is modified by Subclause 11.20, "<rollback statement>", in ISO/IEC 9075-10.

Function

Terminate the current SQL-transaction with rollback, or rollback all actions affecting SQL-data and/or schemas since the establishment of a savepoint.

Format

```
<rollback statement> ::=
  ROLLBACK [ WORK ] [ AND [ NO ] CHAIN ] [ <savepoint clause> ]

<savepoint clause> ::=
  TO SAVEPOINT <savepoint specifier>
```

Syntax Rules

1) If AND CHAIN is specified, then <savepoint clause> shall not be specified.

2) If neither AND CHAIN nor AND NO CHAIN is specified, then AND NO CHAIN is implicit.

Access Rules

None.

General Rules

1) If the current SQL-transaction is part of an encompassing transaction that is controlled by an agent other than the SQL-agent and the <rollback statement> is not being implicitly executed, then an exception condition is raised: *invalid transaction termination*.

2) If a <savepoint clause> is not specified, then:

   a) If an atomic execution context is active, then an exception condition is raised: *invalid transaction termination*.

   b) All changes to SQL-data or schemas that were made by the current SQL-transaction are canceled.

   c) All savepoint levels are destroyed and a new savepoint level is established.

      NOTE 427 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

   d) Every valid locator is marked invalid.

   e) For every open cursor *CR* in the stack of contexts of the current SQL-session, the General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

   f) The current SQL-transaction is terminated.

g) Case:

i) If <rollback statement> contains AND CHAIN, then an SQL-transaction is initiated. Any branch transactions of the SQL-transaction are initiated with the same access mode, isolation level, and diagnostics area limit as the corresponding branch of the SQL-transaction just terminated.

ii) Otherwise:

1) The current access mode, current isolation level, and current condition area limit of the current SQL-session context are set to the access mode, isolation level, and condition area limit, respectively, of the enduring transaction characteristics of the current SQL-session.

2) For every constraint *C*, the constraint mode of *C* in the current SQL-session context is set to the initial constraint mode included in the constraint descriptor for *C*.

3) If a <savepoint clause> is specified, then:

a) Let *S* be the <savepoint name>.

b) If *S* does not specify a savepoint established within the current savepoint level, then an exception condition is raised: *savepoint exception — invalid specification*.

c) If an atomic execution context is active, and *S* specifies a savepoint established before the beginning of the most recent atomic execution context, then an exception condition is raised: *savepoint exception — invalid specification*.

d) All changes to SQL-data or schemas that were made by the current SQL-transaction subsequent to the establishment of *S* are canceled.

e) All savepoints established by the current SQL-transaction subsequent to the establishment of *S* are destroyed.

NOTE 428 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

f) Every valid locator that was generated in the current SQL-transaction subsequent to the establishment of *S* is marked invalid.

g) For every open cursor *CR* that is not a holdable cursor in the stack of contexts of the current SQL-session, the General Rules of Subclause 15.4, “Effect of closing a cursor”, are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

h) The status of any open cursors in the stack of contexts of the current SQL-session that were opened by the current SQL-transaction before the establishment of *S* is implementation-defined.

NOTE 429 — The current SQL-transaction is not terminated, and there is no other effect on the SQL-data or schemas.

Conformance Rules

1) Without Feature T271, “Savepoints”, conforming SQL language shall not contain a <savepoint clause>.

2) Without Feature T261, “Chained transactions”, conforming SQL language shall not contain a <rollback statement> that immediately contains CHAIN.

18 Connection management

18.1 <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 430 — 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.

   c) The access mode, isolation level, and condition area limit of the enduring transaction characteristics of the current SQL-session are read-write, SERIALIZABLE, and an implementation-dependent value not less than 1 (one), respectively.

   NOTE 431 — 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) The current SQL-session context of the current SQL-session is initialized as follows:

a) The authorization stack is set to a single cell containing the user identifier <connection user name>.

b) The current access mode, current isolation level, and current condition area limit are set to the access mode, isolation level, and condition area limit, respectively, of the enduring transaction characteristics of the current SQL-session.

14) A new savepoint level is established.

Conformance Rules

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

18.2 <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 432 — 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 433 — 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>.

18.3 <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 434 — 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>.

(Blank page)

19 Session management

This Clause is modified by Clause 16, "Session management", in ISO/IEC 9075-9.
This Clause is modified by Clause 16, "Session management", in ISO/IEC 9075-14.

19.1 <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> ::=
  <session transaction characteristics>

<session transaction characteristics> ::=
  TRANSACTION <transaction mode> [ { <comma> <transaction mode> }... ]
```

Syntax Rules

1) <session characteristic characteristics> shall contain at most one <isolation level>, at most one <transaction access mode>, and at most one <diagnostics size>.

Access Rules

None.

General Rules

1) Let *SCL* be the <session transaction characteristics>. Let *ESC* be the enduring session characteristics of the current SQL-session.

2) If *SCL* contains an <isolation level> *IL*, then the isolation level of *ESC* is set to the <level of isolation> contained in *IL*.

3) If *SCL* contains a <transaction access mode> *AM*, then the access mode of *ESC* is set to read-only or read-write, according to whether *AM* contains READ ONLY or READ WRITE, respectively.

4) If *SCL* contains a <diagnostics size> *DS*, then the condition area limit of *ESC* is set to the <number of conditions> contained in *DS*.

Conformance Rules

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

19.2 <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) If *V* is not equal to the current value of the SQL-session user identifier of the current SQL-session context, then then restrictions on the permissible values for *V* are implementation-defined.

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>.

19.3 <set role statement>

Function

Set the SQL-session role name and 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) If there is no current user identifier, then an exception condition is raised: *invalid role specification*.

3) If <role specification> contains a <value specification>, then:

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

      `TRIM ( BOTH ' ' FROM S )`

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

   c) 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*.

   d) The SQL-session role name and the current role name are set to *V*.

4) If NONE is specified, then the current role name is removed.

Conformance Rules

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

19.4 <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>.

19.5 <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>.

19.6 <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>.

19.7 <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 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>.

19.8 <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 435 — 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>.

19.9 <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 436 — 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>.

19.10 <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>.

(Blank page)

20 Dynamic SQL

This Clause is modified by Clause 15, "Dynamic SQL", in ISO/IEC 9075-4.
This Clause is modified by Clause 17, "Dynamic SQL", in ISO/IEC 9075-9.
This Clause is modified by Clause 17, "Dynamic SQL", in ISO/IEC 9075-14.

20.1 Description of SQL descriptor areas

This Subclause is modified by Subclause 17.1, "Description of SQL descriptor areas", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.1, "Description of SQL descriptor areas", in ISO/IEC 9075-14.

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, BINARY VARYING, or BINARY LARGE OBJECT and LENGTH is a valid length value for the 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 and *T* is specified by BINARY(*L*), where *L* is the value of LENGTH.

e) Either TYPE indicates BINARY VARYING and *T* is specified by BINARY VARYING or 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.

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

g) 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.

h) 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.

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

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

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

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

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

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

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

p) 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*.

q) 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.

r) 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*.

s) 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.

t) 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.

u) 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*.

v) 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*.

w) 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.

x) 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)

<key word>	Data Type
DATETIME_INTERVAL_PRECI-SION	exact numeric with scale 0 (zero)
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

<key word>	Data Type
SCOPE_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
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 437 — "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
BINARY	60
BINARY VARYING	61
BINARY LARGE OBJECT	30

Data Type	Code
BINARY LARGE OBJECT LOCATOR	31
BOOLEAN	16
CHARACTER	1 (one)
CHARACTER VARYING	12
CHARACTER LARGE OBJECT	40
CHARACTER LARGE OBJECT 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)
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 32, "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 32, "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.

20.2 <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 )`

   Case:

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

   b) Otherwise, let *DN* be the <descriptor name>. The value of *DN* is *V*.

2) Case:

   a) If *DN* identifies an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name*.

   b) Otherwise, an SQL descriptor area is created that is identified by *DN*. 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 value of every other field in the SQL descriptor area is implementation-dependent.

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>.

20.3 <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 <descriptor name> does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name*.

   b) Otherwise, the SQL descriptor area identified by <descriptor name> is destroyed.

Conformance Rules

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

20.4 <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> does not identify an SQL descriptor area, 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 438 — "Match" and "represented by" are defined in the Syntax Rules of Subclause 20.1, "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.

20.5 <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> does not identify an SQL descriptor area, 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 439 — "Match" is defined in the Syntax Rules of Subclause 20.1, "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, BINARY VARYING, or 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>.

20.6 <prepare statement>

This Subclause is modified by Subclause 15.1, "<prepare statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 17.2, "<prepare statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.4, "<prepare statement>", in ISO/IEC 9075-14.

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

   Case:

   a) If *P* contains a <preparable dynamic cursor name> that is ambiguous, then an exception condition is raised: *ambiguous cursor name*.

   b) If *P* contains a <preparable dynamic cursor name> that is invalid, then an exception condition is raised: *invalid cursor name*.

   c) Otherwise, 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 length of varying-length character strings. 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 14.14, "<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, "Result of data type combinations", 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, "Result of data type combinations", 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 *DT* be the simply contained <cast target> of *CS*.

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, "Result of data type combinations", 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, "Result of data type combinations", 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, "Result of data type combinations", 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, "Result of data type combinations", 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, "Result of data type combinations", 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, "Result of data type combinations", 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, "Result of data type combinations", 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) The Syntax Rules of Subclause 10.4, "<routine invocation>", are invoked with a <routine invocation> *RI* as *ROUTINE INVOCATION*, yielding subject routine *SR*.

xxxi) If *DP* is an <SQL argument> of *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 *SR*.

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*.

xxxii) 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*.

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

xxxiv) If *DP* is an <XQuery pattern>, <XQuery option flag>, <regex subject string>, or <XQuery replacement string> immediately contained in a <regex occurrences function>, <regex position expression>, <regex substring function>, <regex transliteration>, or <regex like predicate> *FUN* and if there exists an <XQuery pattern>, <XQuery option flag>, <regex subject string>, or <XQuery replacement string> immediately contained in *FUN*, then let *CS* be the character set of the <XQuery pattern>, <XQuery option flag>, <regex subject string>, or <XQuery replacement string> that is not a <dynamic parameter specification>. *DT* is CHARACTER VARYING (*ML*) CHARACTER SET *CS*.

xxxv) If *DP* is a <start position>, <regex occurrence>, or <regex capture group> immediately contained in a <regex occurrences function>, <regex position expression>, <regex substring function>, or <regex transliteration>, then *DT* is NUMERIC(*MP*).

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) $\leq y \leq n$, 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) $\leq k \leq n$:

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 <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 )
```

Case:

a) If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid SQL statement identifier*.

b) Otherwise, let *ESN* be the <extended statement name>. The value of *ESN* is *V*.

9) 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*.

10) If <SQL statement name> identifies a prepared statement *PS*, then an implicit

```
DEALLOCATE PREPARE SSN
```

is executed, where *SSN* is an <SQL statement name> that identifies *PS*.

11) *P* is prepared for execution, resulting in a prepared statement *PRP*.

Case:

a) If the <prepare statement> is contained in an <SQL-invoked 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 440 — 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.

12) If <attributes specification> is specified, then let *ATV* be the contents of the <attributes variable>. If *ATV* is not a zero-length character string and if *ATV* does not conform to the Format and Syntax Rules of Subclause 20.7, "<cursor attributes>", then an exception condition is raised: *syntax error or access rule violation*.

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>.

20.7 <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.

20.8 <deallocate prepared statement>

This Subclause is modified by Subclause 17.3, "<deallocate prepared statement>", in ISO/IEC 9075-9.

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

   Case:

   a) If the <deallocate prepared statement> is contained in an <SQL-invoked routine>, then the innermost containing <SQL-invoked routine> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <deallocate prepared statement>.

   b) Otherwise, 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 <SQL statement name> does not identify a prepared statement, then an exception condition is raised: *invalid SQL statement name*.

2) If <SQL statement name> identifies a 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. The cursor instance descriptor of any declared dynamic cursor that is associated with the prepared statement is destroyed. The cursor declaration descriptor and cursor instance descriptor of any extended statement cursor 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>.

20.9 <describe statement>

This Subclause is modified by Subclause 17.4, "<describe statement>", in ISO/IEC 9075-9.

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

   Case:

   a) If the <describe statement> is contained in an <SQL-invoked routine>, then the innermost containing <SQL-invoked routine> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <describe statement>.

   b) Otherwise, 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 <SQL statement name> is specified and does not identify a prepared statement, then an exception condition is *invalid SQL statement name*.

2) If <extended cursor name> is specified and does not identify an extended dynamic cursor, then an exception condition is raised: *invalid cursor name*.

3) If <descriptor name> does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name*.

4) Let *DA* be the descriptor area identified by <descriptor name>. Let *N* be the <occurrences> specified when *DA* was allocated.

5) 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).

6) *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 32, "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>, then

      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 441 — 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.

7) 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 442 — 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 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 443 — 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>.

20.10 <input using clause>

This Subclause is modified by , in ISO/IEC 9075-4.
This Subclause is modified by Subclause 17.5, "<input using clause>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.2, "<input using clause>", in ISO/IEC 9075-14.

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>, <SQL parameter reference>, or an <embedded variable specification>.

Access Rules

None.

General Rules

1) If <descriptor name> does not identify an SQL descriptor area, 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 20.1, "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 444 — See the General Rules of Subclause 20.9, "<describe statement>".

      NOTE 445 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "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 446 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "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 ( SV 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 ( 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) The <cast specification>

```
CAST ( SV 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 9.21, "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>.

20.11 <output using clause>

This Subclause is modified by , in ISO/IEC 9075-4.
This Subclause is modified by Subclause 17.6, "<output using clause>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.3, "<output using clause>", in ISO/IEC 9075-14.

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>, <SQL parameter reference>, or an <embedded variable specification>.

Access Rules

None.

General Rules

1) If <descriptor name> does not identify an SQL descriptor area, 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 20.1, "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 447 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "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 448 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "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 *CR* be the cursor identified by the <dynamic fetch statement>, and let *LTDT* be the most specific type of the *i*-th <target specification> or <into argument> of the most recently executed <dynamic fetch statement> prior to the current execution, 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) 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 9.19, "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, CHARACTER LARGE OBJECT, BINARY 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, CHARACTER LARGE OBJECT, BINARY VARYING, 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 449 — 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>.

20.12 <execute statement>

This Subclause is modified by Subclause 17.7, "<execute statement>", in ISO/IEC 9075-9.

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

   Case:

   a) If the <execute statement> is contained in an <SQL-invoked routine>, then the innermost containing <SQL-invoked routine> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <execute statement>.

   b) Otherwise, 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) 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>, then:

   a) If *PS* 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*.

   b) Otherwise, *PS* is treated as a <dynamic single row select statement>.

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 20.10, "<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 a <result using clause> is specified, then the General Rules specified in Subclause 20.11, "<output using clause>", for a <result using clause> in an <execute statement> are applied.

11) 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>.

20.13 <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-dependent <statement name> that does not identify any existing prepared statement.

   NOTE 450 — 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>.

20.14 <dynamic declare cursor>

This Subclause is modified by Subclause 17.8, "<dynamic declare cursor>", in ISO/IEC 9075-9.

Function

Declare a declared dynamic 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 properties>
      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> *M*. The scope of the <cursor name> is *M* with the exception of any <SQL schema statement> contained in *M*.

2) Let *SN* be the <statement name> simply contained in the <dynamic declare cursor>. The containing <SQL-client module definition> shall contain, without an intervening <SQL schema statement>, a <prepare statement> whose <statement name> is equivalent to *SN*.

Access Rules

None.

General Rules

1) A cursor declaration descriptor *CDD* is created. *CDD* includes indications that:

   a) The kind of cursor is a declared dynamic cursor.

   b) The provenance of the cursor is an indication of the SQL-client module whose <SQL-client module definition> contains the <dynamic declare cursor>.

   c) The name of the cursor is the <cursor name>.

   d) The cursor's origin is *SN*.

   e) The cursor's declared properties are as determined by the <cursor properties>.

Conformance Rules

1) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic declare cursor>.

20.15 <allocate cursor statement>

This Subclause is modified by Subclause 17.9, "<allocate cursor statement>", in ISO/IEC 9075-9.

Function

Define a cursor based on a prepared statement for a <cursor specification> or assign a cursor to the result set sequence 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 properties>
      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 )`

   Case:

   a) If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid cursor name*.

   b) Otherwise, let *ECN* be the <extended statement name>. The value of *ECN* is *V*.

2) If *ECN* identifies a cursor, then an exception condition is raised: *invalid cursor name*.

3) If <statement cursor> is specified, then:

   a) If <extended statement name> does not identify a prepared statement, then an exception condition is raised: *invalid SQL statement name*.

   b) If the prepared statement *P* identified by <extended statement name> is not a <cursor specification>, then an exception condition is raised: *dynamic SQL error — prepared statement not a cursor specification*.

   c) *P* is re-prepared as follows: if *P* does not conform to the Syntax Rules of Subclause 14.3, "<cursor specification>", then an exception condition is raised: *syntax error or access rule violation*.

   d) A cursor declaration descriptor *CDD* is created. *CDD* includes:

      i) The kind of cursor is an extended statement cursor.

      ii) The provenance of the cursor is

         Case:

         1) If the <scope option> contained in <extended cursor name> is GLOBAL, then the current SQL-session identifier.

         2) Otherwise, an indication of the SQL-client module whose <SQL-client module definition> contains the <allocate cursor statement>.

      iii) The name of the cursor is the extended name *V* and the explicit or implicit <scope option> of the <extended cursor name>.

      iv) There is no cursor origin.

      v) The cursor's declared properties are as determined by the <cursor properties>.

   e) A cursor instance descriptor *CID* is created. *CID* includes:

      i) The cursor declaration descriptor is *CDD*.

      ii) The SQL-session identifier is the current SQL-session identifier.

      iii) The cursor's state is closed.

   f) 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 cursor declaration descriptor and the cursor instance descriptor of the cursor identified by <extended cursor name> are also destroyed.

4) If <result set cursor> is specified, then:

   a) Let *SIP* be the SQL-invoked procedure identified by <specific routine designator>. Let *INV* be the active SQL-invoked routine of the current routine execution context.

   b) If the SQL-session context of the current SQL-session does not include a result set sequence *RSS* brought into existence by an invocation of *SIP* by *INV*, then an exception condition is raised: *invalid SQL-invoked procedure reference*.

   c) If *RSS* is empty, then an exception condition is raised: *no data — no additional result sets returned*.

   d) A cursor declaration descriptor *CDD* is created. *CDD* includes indications that:

i) The kind of cursor is a received cursor.

ii) The provenance of the cursor is

Case:

1) If the <scope option> contained in <extended cursor name> is GLOBAL, then the current SQL-session identifier.

2) Otherwise, an indication of the SQL-client module whose <SQL-client module definition> contains the <allocate cursor statement>.

iii) The name of the cursor is the extended name *V* and the explicit or implicit <scope option> of the <extended cursor name>.

iv) The cursor origin is the <specific routine designator>.

v) The cursor's declared properties are as follows:

1) The cursor's declared sensitivity is ASENSITIVE.

2) The cursor's declared scrollability is NO SCROLL.

3) The cursor's declared holdability is WITHOUT HOLD.

4) The cursor's declared returnability is WITHOUT RETURN.

e) A cursor instance descriptor *CID* is created. *CID* includes:

i) The cursor declaration descriptor is *CDD*.

ii) The current SQL-session identifier.

iii) The cursor's state is open.

f) The General Rules of Subclause 15.2, “Effect of advancing a received cursor to the next result set”, are applied, with *CID* as *CURSOR*.

Conformance Rules

1) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain an <allocate cursor statement>.

2) Without Feature T471, “Result sets return value”, conforming SQL language shall not contain a <result set cursor>.

20.16 <dynamic open statement>

This Subclause is modified by Subclause 17.10, "<dynamic open statement>", in ISO/IEC 9075-9.

Function

Associate input dynamic parameters with a <cursor specification> and open the dynamic 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 *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor. Let *CDD* be the cursor declaration descriptor identified by *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) Case:

   a) If *DCN* is a <cursor name> and the <statement name> contained in *CDD* does not identify a prepared statement that is a <cursor specification>, then an exception condition is raised: *invalid SQL statement name*.

   b) Otherwise, if *DCN* does not identify an extended dynamic cursor, then an exception condition is raised: *invalid cursor name*.

2) Let *CR* be the cursor identified by *DCN*.

3) If the prepared statement *P* 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) If the <dynamic cursor name> is a <cursor name>, then *P* is re-prepared as follows: if *P* does not conform to the Syntax Rules of Subclause 14.3, "<cursor specification>", then an exception condition is raised: *syntax error or access rule violation*.

5) *CR* is updatable if and only if the <cursor specification> included in the result set descriptor of *CR* is updatable.

6) If an <input using clause> is specified, then the General Rules specified in Subclause 20.10, "<input using clause>", for <dynamic open statement> are applied.

7) The General Rules of Subclause 15.1, "Effect of opening a cursor", are applied with *CR* as *CURSOR*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic open statement>.

20.17 <dynamic fetch statement>

This Subclause is modified by Subclause 17.11, "<dynamic fetch statement>", in ISO/IEC 9075-9.

Function

Fetch a row for a dynamic 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 *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

Access Rules

None.

General Rules

1) If *DCN* does not identify a cursor, then an exception condition is raised: *invalid cursor name*.

2) Let *CR* be the cursor identified by *DCN*.

3) The General Rules of Subclause 15.3, "Determination of the current row of a cursor", are applied with *CR* as *CURSOR* and <fetch orientation> as *FETCH ORIENTATION*.

4) If a completion condition *no data* is raised, then no further General Rules of this Subclause are applied.

5) The General Rules specified in Subclause 20.11, "<output using clause>", for a <result using clause> in a <dynamic fetch statement> are applied.

6) If an exception condition is raised during the assignment of a value to a target, then the values of all targets are implementation-dependent and *CR* remains positioned on the current row.

   NOTE 451 — It is implementation-dependent whether *CR* remains positioned on the current row when an exception condition is raised during the derivation of any <derived column>.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic fetch statement>.

20.18 <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>.

20.19 <dynamic close statement>

This Subclause is modified by Subclause 17.12, "<dynamic close statement>", in ISO/IEC 9075-9.

Function

Close a dynamic cursor.

Format

```
<dynamic close statement> ::=
  CLOSE <dynamic cursor name>
```

Syntax Rules

1) If <dynamic cursor name> *DCN* is a <cursor name> *CN*, then *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

Access Rules

None.

General Rules

1) If *DCN* does not identify a cursor, then an exception condition is raised: *invalid cursor name*.

2) Let *CR* be the cursor identified by *DCN*.

3) The General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic close statement>.

20.20 <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 *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

2) Let *TN* be the <table name> contained in <target table>.

Access Rules

1) Case:

   a) If <dynamic delete statement: positioned> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges for the owner of that schema shall include DELETE for *TN*.

   b) Otherwise, the current privileges shall include DELETE for *TN*.

General Rules

1) If *DCN* does not identify a dynamic cursor, then an exception condition is raised: *invalid cursor name*.

2) Let *CR* be the cursor identified by *DCN*.

3) If *CR* is not an updatable cursor, then an exception condition is raised: *invalid cursor name*.

4) Let *T* be the simply underlying table of *CR*. *T* is the subject table of the <dynamic delete statement: positioned>. Let *LUT* be the leaf underlying table of *T* such that *T* is one-to-one with *LUT*. Let *LUTN* be a <table name> that identifies *LUT*.

5) 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*.

6) The General Rules of Subclause 15.5, “Effect of a positioned delete”, are applied with *CR* as *CURSOR*.

Conformance Rules

1) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic delete statement: positioned>.

20.21 <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 *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

2) Let *TN* be the <table name> contained in <target table>.

3) The scope of *TN* is the entire <dynamic update statement: positioned>.

Access Rules

1) Case:

   a) If <dynamic update statement: positioned> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* shall include UPDATE for each <object column>.

   b) Otherwise, the current privileges shall include UPDATE for each <object column>.

General Rules

1) If *DCN* does not identify a dynamic cursor, then an exception condition is raised: *invalid cursor name*.

2) Let *CR* be the cursor identified by *DCN*.

3) If *CR* is not an updatable cursor, then an exception condition is raised: *invalid cursor name*.

4) Let *T* be the simply underlying table of *CR*. *T* is the subject table of the <dynamic update statement: positioned>. Let *LUT* be the leaf underlying table of *T* such that *T* is one-to-one with *LUT*. Let *LUTN* be a <table name> that identifies *LUT*.

5) 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*.

6) 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*.

7) 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> included in the result set descriptor of *CR*, then an exception condition is raised: *attempt to assign to non-updatable column*.

8) Let *SCL* be the <set clause list>.

9) The General Rules of Subclause 15.6, "Effect of a positioned update", are applied with *CR* as *CURSOR* and *SCL* as *SET CLAUSE LIST*. 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>.

20.22 <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 <preparable dynamic cursor name>
```

Syntax Rules

1) If the <preparable dynamic cursor name> *PDCN* is not ambiguous or invalid, then:

   a) Let *CR* be the cursor identified by *PDCN*.

   b) If <target table> is not specified, then let *QE* be the <query expression> simply contained in the <cursor specification> of the result set descriptor of *CR*. Let *LUT* be the leaf underlying table of *QE* such that *QE* is one-to-one with respect to *QE*. Let *TN* be the name of *LUT*.

      Case:

      i) If the <table reference> that references *LUT* specifies ONLY, then the <target table>

         `ONLY ( TN )`

         is implicit.

      ii) Otherwise, the <target table>

         TN

         is implicit.

   c) Let *TT* be the explicit or implicit <target table>.

   d) Let *TN* be the <table name> contained in *TT*.

   e) If *TT* immediately contains ONLY and the table identified by *TN* is not a typed table, then *TT* is equivalent to *TN*.

   f) Let *T* be the simply underlying table of *CDD*. *T* is the *subject table* of the <preparable dynamic delete statement: positioned>. Let *LUT* be the leaf underlying table of *T* such that *T* is one-to-one with respect to *LUT*.

   g) The subject table of a <preparable dynamic delete statement: positioned> shall not identify an old transition table or a new transition table.

   h) *TN* shall identify *LUT*.

   i) <target table> shall specify ONLY if and only if the <table reference> contained in *T* that references *LUT* specifies ONLY.

j) The schema identified by the explicit or implicit qualifier of *TN* shall include the descriptor of *LUT*.

Access Rules

1) If *PDCN* is not ambiguous or invalid, then

Case:

a) If <preparable dynamic delete statement: positioned> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges for the owner of that schema shall include DELETE for *TN*.

b) Otherwise, the current privileges shall include DELETE for *TN*.

General Rules

1) The General Rules of Subclause 15.5, "Effect of a positioned delete", are applied with *CR* as *CURSOR*.

NOTE 452 — If the General Rules are reached, then *PDCN* cannot be ambiguous or invalid.

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <preparable dynamic delete statement: positioned>.

20.23 <preparable dynamic cursor name>

Function

Specify the cursor of a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>.

Format

```
<preparable dynamic cursor name> ::=
  [ <scope option> ] <cursor name>
```

Syntax Rules

1) Let *PDCN* be the <preparable dynamic cursor name>, and let *CN* be the <cursor name> contained in *PDCN*.

   Case:

   a) If *PDCN* contains a <scope option> that specifies GLOBAL, then

      Case:

      i) If there exists in the current SQL-session context 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 *PDCN*.

      ii) Otherwise, *PDCN* is said to be *invalid*.

   b) If *PDCN* contains a <scope option> that specifies LOCAL, or if no <scope option> is specified, then the *potentially referenced cursors* of *PDCN* include every declared dynamic cursor whose <cursor name> is equivalent to *CN* and whose scope is the containing SQL-client module (minus any <SQL schema statement>s contained in the SQL-client module) and every extended dynamic cursor having an <extended cursor name> that has a scope of the containing SLI-client module (minus any <SQL schema statement>s contained in the SQL-client module) and whose <cursor name> is equivalent to *CN*.

      Case:

      i) If the number of potentially referenced cursors is greater than 1 (one), then *PDCN* is said to be *ambiguous*.

      ii) If the number of potentially referenced cursors is less than 1 (one), then *PDCN* is said to be *invalid*.

      iii) Otherwise, *PDCN* refers to the single potentially referenced cursor of *P*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

20.24 <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 <preparable dynamic cursor name>
```

Syntax Rules

1) If the <preparable dynamic cursor name> *PDCN* is not ambiguous or invalid, then:

   a) Let *CR* be the cursor identified by *PDCN*.

   b) If <target table> is not specified, then let *QE* be the <query expression> simply contained in the <cursor specification> of the result set descriptor of the cursor identified by <preparable dynamic cursor name>. Let *LUT* be the leaf underlying table of *QE* such that *QE* is one-to-one with respect to *LUT*. Let *TN* be the name of *LUT*.

      Case:

      i) If the <table reference> that references *LUT* specifies ONLY, then the <target table>

         ONLY (*TN*)

         is implicit.

      ii) Otherwise, the <target table>

         TN

         is implicit.

   c) Let *TT* be the explicit or implicit <target table>.

   d) Let *TU* be the simply underlying table of *CR*. *TU* is the *subject table* of the <update statement: positioned>. Let *LUT* be the leaf underlying table *T* such that *T* is one-to-one with respect to *LUT*.

   e) Let *TN* be the <table name> contained in *TT*. *TN* shall identify *LUT*.

   f) *TT* shall specify ONLY if and only if the <table reference> contained in *TU* that references *LUT* specifies ONLY.

   g) *TN* shall not identify an old transition table or a new transition table.

   h) Let *T* be the table identified by *TN*.

   i) Let *CN* be the <table name> contained in <target table>. *CN* is an exposed <table or query name>.

   j) The scope of *CN* is <set clause list>.

k) If *CR* is an ordered cursor, then for each <object column> *OC* contained in <set clause list>, the <order by clause> of the <cursor specification> for *CR* shall not generally contain a <column reference> that references *OC* or an underlying column of the column identified by *OC*.

l) Each <column name> specified as an <object column> shall identify a column in the explicit or implicit <column name list> contained in the explicit or implicit <updatability clause> of the <cursor specification> included in the result set descriptor of *CR*.

Access Rules

1) If *PDCN*is not ambiguous or invalid, then

   Case:

   a) If <preparable dynamic update statement: positioned> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* shall include UPDATE for each <object column>.

   b) Otherwise, the current privileges shall include UPDATE for each <object column>.

General Rules

1) Let *SCL* be the <set clause list>.

2) The General Rules of Subclause 15.6, "Effect of a positioned update", are applied with *CR* as *CURSOR* and *SCL* as *SET CLAUSE LIST*.

   NOTE 453 — If the General Rules are reached, then *PDCN* cannot be ambiguous or invalid.

Conformance Rules

1) Without Feature B032, "Extended dynamic SQL", conforming SQL language shall not contain a <preparable dynamic update statement: positioned>.

21 Embedded SQL

This Clause is modified by Clause 16, "Embedded SQL", in ISO/IEC 9075-4.
This Clause is modified by Clause 18, "Embedded SQL", in ISO/IEC 9075-9.
This Clause is modified by Clause 8, "Embedded SQL", in ISO/IEC 9075-10.
This Clause is modified by Clause 18, "Embedded SQL", in ISO/IEC 9075-14.

21.1 <embedded SQL host program>

This Subclause is modified by Subclause 16.1, "<embedded SQL host program>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 8.1, "<embedded SQL host program>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 18.1, "<embedded SQL host program>", in ISO/IEC 9075-14.

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 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 454 — "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 455 — 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 21.3, "<embedded SQL Ada program>", Subclause 21.4, "<embedded SQL C program>", Subclause 21.5, "<embedded SQL COBOL program>", Subclause 21.6, "<embedded SQL Fortran program>", Subclause 21.7, "<embedded SQL MUMPS program>", Subclause 21.8, "<embedded SQL Pascal program>", and Subclause 21.9, "<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 456 — 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 9.21, "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) ≤ j ≤ a, be the host variable corresponding to PI_j. Let HVO_k, 1 (one) ≤ k ≤ b, be the host variable corresponding to PO_k. Let $HVIO_l$, 1 (one) ≤ l ≤ c, be the host variable corresponding to PIO_l.

E) Let TSI_j, 1 (one) ≤ j ≤ a, be the associated SQL data type of HVI_j. Let TSO_k, 1 (one) ≤ k ≤ b, be the associated SQL data type of HVO_k. Let $TSIO_l$, 1 (one) ≤ l ≤ c, be the associated SQL data type of $HVIO_l$.

F) Let TUI_j, 1 (one) ≤ j ≤ a, be the associated user-defined type of HVI_j. Let TUO_k, 1 (one) ≤ k ≤ b, be the associated user-defined type of HVO_k. Let $TUIO_l$, 1 (one) ≤ l ≤ c, be the associated user-defined type of $HVIO_l$.

G) Let GNI_j, 1 (one) ≤ j ≤ 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) ≤ k ≤ 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) ≤ l ≤ 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) ≤ j ≤ a, apply the Syntax Rules of Subclause 9.21, "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) ≤ l ≤ c, apply the Syntax Rules of Subclause 9.21, "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) ≤ k ≤ b, apply the Syntax Rules of Subclause 9.19, "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) ≤ l ≤ c, apply the Syntax Rules of Subclause 9.19, "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) ≤ j ≤ a, SVO_k, 1 (one) ≤ k ≤ b, and $SVIO_l$, 1 (one) ≤ l ≤ 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) ≤ i ≤ 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) ≤ j ≤ 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) ≤ k ≤ 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) ≤ l ≤ 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 21.2, "<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 457 — In an <embedded SQL Fortran program>, the "SQLSTATE" variable may be abbreviated to "SQLSTA". See the Syntax Rules of Subclause 21.6, "<embedded SQL Fortran program>".

ii) Exception actions, as specified in Subclause 21.2, "<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>.

21.2 <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 33, "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 33, "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 458 —

   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 33, "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 33, "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 33, "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>.

21.3 <embedded SQL Ada program>

This Subclause is modified by Subclause 18.1, "<embedded SQL Ada program>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 18.2, "<embedded SQL Ada program>", in ISO/IEC 9075-14.

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.CHAR
        [ CHARACTER SET [ IS ] <character set specification> ]
        <left paren> 1 <double period> <length> <right paren>
  | Interfaces.SQL.SMALLINT
  | Interfaces.SQL.INT
  | Interfaces.SQL.BIGINT
  | Interfaces.SQL.REAL
  | Interfaces.SQL.DOUBLE_PRECISION
  | Interfaces.SQL.BOOLEAN
  | Interfaces.SQL.SQLSTATE_TYPE
  | Interfaces.SQL.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 BINARY variable>
  | <Ada VARBINARY 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 BINARY variable> ::=
  SQL TYPE IS BINARY <left paren> <length> <right paren>

<Ada VARBINARY variable> ::=
  SQL TYPE IS VARBINARY <left paren> <length> <right paren>

<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 21.1, "<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 BINARY (L)
```

   for a given <Ada host identifier> *HVN* shall be replaced by

```
HVN : Interfaces.SQL.CHAR(1..L)
```

   in any <Ada BINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

   d) The syntax

```
SQL TYPE IS VARBINARY (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);
```

   in any <Ada VARBINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

   e) 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>".

f) 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.

g) 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*.

h) 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*.

i) 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.

j) 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.

k) 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.

l) 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 21.1, "<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 T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain an <Ada BINARY variable>.

9) Without Feature T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain an <Ada VARBINARY variable>.

10) Without Feature S241, "Transform functions", conforming SQL language shall not contain an <Ada user-defined type variable>.

11) Without Feature S041, "Basic reference types", conforming SQL language shall not contain an <Ada REF variable>.

12) Without Feature S232, "Array locators", conforming SQL language shall not contain an <Ada array locator variable>.

13) Without Feature S233, "Multiset locators", conforming SQL language shall not contain an <Ada multiset locator variable>.

14) 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.

21.4 <embedded SQL C program>

This Subclause is modified by Subclause 18.2, "<embedded SQL C program>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 18.3, "<embedded SQL C program>", in ISO/IEC 9075-14.

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 BINARY variable>
  | <C VARBINARY 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 BINARY variable> ::=
  SQL TYPE IS BINARY <left paren> <length> <right paren>
      <C host identifier> [ <C initial value> ]
      [ { <comma> <C host identifier> [ <C initial value> ] }... ]

<C VARBINARY variable> ::=
  SQL TYPE IS VARBINARY <left paren> <length> <right paren>
```

```
      <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 21.1, "<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 459 — 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 BINARY (L)
```

for a given <C host identifier> *hvn* shall be replaced by

```
unsigned char HVN [L]
```

in any <C BINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

h) The syntax

```
SQL TYPE IS VARBINARY (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 VARBINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

i) 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.

j) 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*.

k) 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*.

l) 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.

m) 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.

n) 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.

o) 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”, “BINARY” “VARBINARY”, “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 21.1, “<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`.

12) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <C BINARY variable>.

13) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <C VARBINARY variable>.

21.5 <embedded SQL COBOL program>

This Subclause is modified by Subclause 18.3, "<embedded SQL COBOL program>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 18.4, "<embedded SQL COBOL program>", in ISO/IEC 9075-14.

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 BINARY 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 BINARY variable> ::=
  [ USAGE [ IS ] ] SQL TYPE IS BINARY <left paren> <length> <right paren>

<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 460 — 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 21.1, "<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 461 — 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 462 — 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 BINARY (L)
```

   shall be replaced by

```
PICTURE X (L)
```

   in any <COBOL BINARY variable>.

   e) 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.

f) 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*.

g) 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*.

h) 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.

i) 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.

j) 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.

k) 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 21.1, “<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>.

11) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <COBOL BINARY variable>.

21.6 <embedded SQL Fortran program>

This Subclause is modified by Subclause 18.4, "<embedded SQL Fortran program>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 18.5, "<embedded SQL Fortran program>", in ISO/IEC 9075-14.

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 BINARY variable>
  | <Fortran VARBINARY 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 BINARY variable> ::=
  SQL TYPE IS BINARY <left paren> <length> <right paren>

<Fortran VARBINARY variable> ::=
  SQL TYPE IS VARBINARY <left paren> <length> <right paren>
```

```
<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 21.1, "<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 463 — 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 BINARY (L)
```

for a given <Fortran host identifier> *HVN* shall be replaced by

```
CHARACTER HVN * L
```

in any <Fortran BINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

e) The syntax

```
SQL TYPE IS VARBINARY (L)
```

for a given <Fortran host identifier> *HVN* shall be replaced by

```
    INTEGER HVN_RESERVED
    INTEGER HVN_LENGTH
    CHARACTER HVN_DATA (* L);
```

in any <Fortran VARBINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

f) 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.

g) 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*.

h) 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*.

i) 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.

j) 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.

k) 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.

l) 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 21.1, "<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>.

11) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <Fortran BINARY variable>.

12) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <Fortran VARBINARY variable>.

21.7 <embedded SQL MUMPS program>

This Subclause is modified by Subclause 18.5, "<embedded SQL MUMPS program>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 18.6, "<embedded SQL MUMPS program>", in ISO/IEC 9075-14.

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 VARBINARY 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 VARBINARY variable> ::=
```

```
  SQL TYPE IS VARBINARY <left paren> <length> <right paren>

<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 21.1, “<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 VARBINARY (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 VARBINARY variable>, where *L* is the numeric value of <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 <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.

e) 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*.

f) 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*.

g) 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.

h) 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.

i) 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.

j) 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 21.1, “<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>.

11) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <MUMPS VARBINARY variable>.

21.8 <embedded SQL Pascal program>

This Subclause is modified by Subclause 18.6, "<embedded SQL Pascal program>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 18.7, "<embedded SQL Pascal program>", in ISO/IEC 9075-14.

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 BINARY 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 BINARY variable> ::=
  SQL TYPE IS BINARY <left paren> <length> <right paren>

<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 21.1, "<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 BINARY (L)
```

for a given <Pascal host identifier> *HVN* shall be replaced by

```
HVN : PACKED ARRAY [1..L] OF CHAR
```

in any <Pascal BINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

f) 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.

g) 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*.

h) 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*.

i) 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.

j) 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.

k) 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.

l) 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 two 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 21.1, “<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>.

11) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <Pascal BINARY variable>.

21.9 <embedded SQL PL/I program>

This Subclause is modified by Subclause 18.7, "<embedded SQL PL/I program>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 18.8, "<embedded SQL PL/I program>", in ISO/IEC 9075-14.

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 BINARY variable>
  | <PL/I VARBINARY 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 BINARY variable> ::=
  SQL TYPE IS BINARY <left paren> <length> <right paren>

<PL/I VARBINARY variable> ::=
  SQL TYPE IS VARBINARY <left paren> <length> <right paren>

<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 21.1, "<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 BINARY (L)
```

for a given <PL/I host identifier> *HVN* shall be replaced by

```
DCL HVN CHARACTER (L)
```

in any <PL/I BINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

e) The syntax

```
SQL TYPE IS VARBINARY (L)
```

for a given <PL/I host identifier> *HVN* shall be replaced by

```
DCL HVN CHARACTER (L) VARYING
```

in any <PL/I VARBINARY variable>, where *L* is the numeric value of <length> as specified in Subclause 5.2, "<token> and <separator>".

f) 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.

g) 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*.

h) 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*.

i) 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.

j) 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.

k) 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.

l) 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 21.1, “<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>.

11) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <PL/I BINARY variable>.

12) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <PL/I VARBINARY variable>.

22 Direct invocation of SQL

22.1 <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) A copy of the top cell of the authorization stack is pushed onto the authorization stack.

4) Let *S* be the <direct SQL statement>.

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 464 — 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 465 — 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>.

22.2 <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_CATALOG, CURRENT_SCHEMA, 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.

(Blank page)

23 Diagnostics management

This Clause is modified by Clause 17, "Diagnostics management", in ISO/IEC 9075-4.
This Clause is modified by Clause 23, "Diagnostics management", in ISO/IEC 9075-9.
This Clause is modified by Clause 19, "Diagnostics management", in ISO/IEC 9075-14.

23.1 <get diagnostics statement>

This Subclause is modified by Subclause 17.1, "<get diagnostics statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 23.1, "<get diagnostics statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 19.1, "<get diagnostics statement>", in ISO/IEC 9075-14.

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>
  | <all 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> ::=
  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

<all information> ::=
  <all info target> <equals operator> ALL [ <all qualifier> ]

<all info target> ::=
  <simple target specification>

<all qualifier> ::=
    STATEMENT
  | CONDITION [ <condition number> ]

<condition number> ::=
  <simple value specification>
```

Syntax Rules

1) Let the <simple target specification> contained in a <statement information item> or <condition information item> be *STS*. Let *IDDT* be the declared type specified in Table 30, "<statement information item name>s for use with <get diagnostics statement>", for the corresponding <statement information item name> or in Table 31, "<condition information item name>s for use with <get diagnostics statement>", for the corresponding <condition information item name>. Let *ID* be the expression:

   ```
   CAST ( '' AS IDDT )
   ```

ID shall be assignable to *STS* according to the Syntax Rules of Subclause 9.1, "Retrieval assignment", with *ID* as *VALUE* and *STS* as *TARGET*.

2) The declared type of <all info target> shall be a character string type.

3) The declared type of <condition number> shall be exact numeric with scale 0 (zero).

Table 30 — <statement information item name>s for use with <get diagnostics statement>

<identifier>	Declared Type
COMMAND_FUNCTION	variable-length character string with implementation-defined maximum length
COMMAND_FUNCTION_CODE	exact numeric with scale 0 (zero)
DYNAMIC_FUNCTION	variable-length character string with implementation-defined maximum length
DYNAMIC_FUNCTION_CODE	exact numeric with scale 0 (zero)
MORE	fixed-length character string with length 1 (one)
NUMBER	exact numeric with scale 0 (zero)
ROW_COUNT	exact numeric with scale 0 (zero)
TRANSACTION_ACTIVE	exact numeric with scale 0 (zero)
TRANSACTIONS_COMMITTED	exact numeric with scale 0 (zero)
TRANSACTIONS_ROLLED_BACK	exact numeric with scale 0 (zero)

Table 31 — <condition information item name>s for use with <get diagnostics statement>

<identifier>	Declared Type
CATALOG_NAME	variable-length character string with implementation-defined maximum length
CLASS_ORIGIN	variable-length character string with implementation-defined maximum length
COLUMN_NAME	variable-length character string with implementation-defined maximum length
CONDITION_NUMBER	exact numeric with scale 0 (zero)

<identifier>	Declared Type
CONNECTION_NAME	variable-length character string with implementation-defined maximum length
CONSTRAINT_CATALOG	variable-length character string with implementation-defined maximum length
CONSTRAINT_NAME	variable-length character string with implementation-defined maximum length
CONSTRAINT_SCHEMA	variable-length character string with implementation-defined maximum length
CURSOR_NAME	variable-length character string with implementation-defined maximum length
MESSAGE_LENGTH	exact numeric with scale 0 (zero)
MESSAGE_OCTET_LENGTH	exact numeric with scale 0 (zero)
MESSAGE_TEXT	variable-length character string with implementation-defined maximum length
PARAMETER_MODE	variable-length character string with maximum length 5
PARAMETER_NAME	variable-length character string with implementation-defined maximum length
PARAMETER_ORDINAL_POSITION	exact numeric with scale 0 (zero)
RETURNED_SQLSTATE	fixed-length character string with length 5
ROUTINE_CATALOG	variable-length character string with implementation-defined maximum length
ROUTINE_NAME	variable-length character string with implementation-defined maximum length
ROUTINE_SCHEMA	variable-length character string with implementation-defined maximum length
SCHEMA_NAME	variable-length character string with implementation-defined maximum length
SERVER_NAME	variable-length character string with implementation-defined maximum length
SPECIFIC_NAME	variable-length character string with implementation-defined maximum length

<identifier>	Declared Type
SUBCLASS_ORIGIN	variable-length character string with implementation-defined maximum length
TABLE_NAME	variable-length character string with implementation-defined maximum length
TRIGGER_CATALOG	variable-length character string with implementation-defined maximum length
TRIGGER_NAME	variable-length character string with implementation-defined maximum length
TRIGGER_SCHEMA	variable-length character string with implementation-defined maximum length

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 466 — 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 32, "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 32, "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 32, "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 32, "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 32 — 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
<deallocate prepared statement>	DEALLOCATE PREPARE	16
<delete statement: positioned>	DELETE CURSOR	18
<delete statement: searched>	DELETE WHERE	19
<describe statement>	DESCRIBE	20

SQL-statement	Identifier	Code
<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
<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

SQL-statement	Identifier	Code
<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
<rollback statement>	ROLLBACK WORK	62
<savepoint statement>	SAVEPOINT	63
<schema definition>	CREATE SCHEMA	64
<schema routine>	CREATE ROUTINE	14

SQL-statement	Identifier	Code
<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
<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

SQL-statement	Identifier	Code
<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 467 — 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 468 — 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 469 — 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 470 — 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 32, "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 24.1, "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 24.1, "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 471 — 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 472 — There are no numeric items that are not set by these rules.

6) Case:

a) If <SQL diagnostics information> contains an <all information>, then let *N* be the number of occupied condition areas in *DA*. Let *S* be the <literal> `''`. For each *X* such that *X* is either the <statement information item name> of a statement information item in *DA*, or the <condition information item name> of a condition information item in an occupied condition area of *DA*, or CONDITION_NUMBER (whose value is the ordinal position of an occupied condition area), let *CHARFORM* (*X*) denote the <value expression>

```
CAST ( LX AS CHARACTER VARYING ( M ) )
```

where *LX* is a <literal> denoting the value of *X* and *M* is the implementation-defined maximum length of variable-length character strings. Let `PRESENTATION ( X )` denote the <value expression>

```
'X = ' || CHARFORM ( X )
```

Let *SEMICOLONCAT* (*str*, *din*) be equivalent to

```
CASE
  WHEN CHARFORM ( din ) = '' THEN str ;
  WHEN str = '' THEN PRESENTATION ( din ) ;
  ELSE str || ';' || PRESENTATION ( din ) ;
END CASE
```

i) If <all qualifier> is omitted or contains STATEMENT, then for each statement information item *SII* in *DA*, taken in an implementation-dependent order, let *S* be

```
SEMICOLONCAT ( S , SIN )
```

where *SIN* is the <statement information item name> of *SII*.

ii) If <all qualifier> is omitted or contains CONDITION and does not contain a <condition number>, then for each *CN*, 1 (one) ≤ *CN* ≤ *N*, in an implementation-defined order:

1) Let *S* be

```
SEMICOLONCAT ( S, COND )
```

where *COND* is CONDITION_NUMBER and the value of CONDITION_NUMBER is *CN*.

2) For each condition information item *CII* in condition area *CN*, taken in an implementation-dependent order, let *S* be

```
SEMICOLONCAT ( S, CIN )
```

where *CIN* is the <condition number> of *CII*.

iii) If <all qualifier> contains CONDITION and a <condition number>, then let *CN* be the value of <condition number>. For each condition information item *CII* in condition area *CN*, taken in an implementation-dependent order, let *S* be

```
SEMICOLONCAT ( S, CIN )
```

where *CIN* is the <condition item number> of *CII*.

iv) The General Rules of Subclause 9.1, "Retrieval assignment", apply to <all info target> and *S*, as *TARGET* and *VALUE*, respectively.

b) If <statement information> is specified, then for each <statement information item>, the General Rules of Subclause 9.1, "Retrieval assignment", are applied with <simple target specification> as *TARGET* and the value of <statement information item name> as *VALUE*.

c) Otherwise (<condition information> is specified), for each <condition information item>, the General Rules of Subclause 9.1, "Retrieval assignment", are applied with <simple target specification> as *TARGET* and the value of <condition information item name> as *VALUE*.

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.

3) Without Feature F122, “Enhanced diagnostics management”, conforming SQL language shall not contain a <get diagnostics statement> containing a <simple target specification> whose declared type is different from that of the corresponding <statement information item name> or <condition information item name>.

4) Without Feature F123, “All diagnostics”, conforming SQL language shall not contain an <all information>.

23.2 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.

24 Status codes

This Clause is modified by Clause 20, "Status codes", in ISO/IEC 9075-4.
This Clause is modified by Clause 26, "Status codes", in ISO/IEC 9075-9.
This Clause is modified by Clause 15, "Status codes", in ISO/IEC 9075-10.
This Clause is modified by Clause 15, "Status codes", in ISO/IEC 9075-13.
This Clause is modified by Clause 23, "Status codes", in ISO/IEC 9075-14.

24.1 SQLSTATE

This Subclause is modified by Subclause 20.1, "SQLSTATE", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 26.1, "SQLSTATE", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 15.1, "SQLSTATE", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 15.1, "SQLSTATE", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 23.1, "SQLSTATE", in ISO/IEC 9075-14.

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 33, "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 473 — 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 33 — 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

Category	Condition	Class	Subcondition	Subclass
X	*data exception*	22	(*no subclass*)	000
			array data, right truncation	02F
			array element error	02E
			attempt to replace a zero-length string	01U
			character not in repertoire	021
			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

Category	Condition	Class	Subcondition	Subclass
			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
			invalid time zone displacement value	009
			invalid use of escape character	00C
			invalid XQuery option flag	01T
			invalid XQuery regular expression	01S
			invalid XQuery replacement string	01V
			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

Category	Condition	Class	Subcondition	Subclass
			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
			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

Category	Condition	Class	Subcondition	Subclass
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
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

Category	Condition	Class	Subcondition	Subclass
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
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 result sets returned	001
X	*prohibited statement encountered during trigger execution*	0W	(*no subclass*)	000

Category	Condition	Class	Subcondition	Subclass
X	Remote Database Access	*HZ*	(*See Table 34, "SQLSTATE class codes for RDA", for the definition of protocol subconditions and subclass code values*)	
X	*savepoint exception*	3B	(*no subclass*)	000
			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

Category	Condition	Class	Subcondition	Subclass
			attempt to return too many result sets	00E
			cursor operation conflict	001
			default value too long for information schema	00B
			disconnect error	002
			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

24.2 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 34, “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 34, “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 34, “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 34, “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 33, “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 34 — SQLSTATE class codes for RDA

SQLSTATE Class	Subclass Origin
HZ	ISO/IEC 9579

25 Conformance

25.1 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 of 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”

25.2 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.

25.3 Implied feature relationships of SQL/Foundation

Table 35 — 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
F053	OVERLAPS predicate	F052	Intervals and date arithmetic
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
F693	SQL-session and client module collations	F690	Collation support
F711	ALTER domain	F251	Domain support
F721	Deferrable constraints	F491	Constraint management
F762	CURRENT_CATALOG	F651	Catalog name qualifiers
F762	CURRENT_CATALOG	F763	CURRENT_SCHEMA
F801	Full set function	F441	Extended set function support
F846	Octet support in regular expression operators	T061	UCS 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

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
S232	Array locators	S091	Basic array support
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
T043	Multiplier T	T041	Basic LOB data type support
T044	Multiplier P	T043	Multiplier T
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
T152	DISTINCT predicate with negation	T151	DISTINT predicate
T173	Extended LIKE clause in table definition	T171	LIKE clause in table definition
T212	Enhanced trigger capability	T211	Basic trigger capability
T272	Enhanced savepoint management	T271	Savepoints
T332	Extended roles	T331	Basic roles
T432	Nested and concatenated GROUPING SETS	T431	Extended grouping capabilities

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
T433	Multiargument GROUPING function	T431	Extended grouping capabilities
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
T652	SQL-dynamic statements in SQL routines	B031	Basic dynamic SQL
T654	SQL-dynamic statements in external routines	B031	Basic dynamic SQL
T022	Advanced BINARY and VARBINARY data type support	T021	BINARY and VARBINARY data types

(Blank page)

Annex A
(informative)

SQL Conformance Summary

This Annex is modified by Annex A, "SQL Conformance Summary", in ISO/IEC 9075-4.
This Annex is modified by Annex A, "SQL Conformance Summary", in ISO/IEC 9075-9.
This Annex is modified by Annex A, "SQL Conformance Summary", in ISO/IEC 9075-10.
This Annex is modified by Annex A, "SQL Conformance Summary", in ISO/IEC 9075-11.
This Annex is modified by Annex A, "SQL Conformance Summary", in ISO/IEC 9075-13.
This Annex is modified by Annex A, "SQL Conformance Summary", in ISO/IEC 9075-14.

The contents of this Annex summarizes all Conformance Rules, ordered by Feature ID and by Subclause.

1) Specifications for Feature B011, "Embedded Ada":

   a) Subclause 21.3, "<embedded SQL Ada program>":

      i) Without Feature B011, "Embedded Ada", conforming SQL language shall not contain an <embedded SQL Ada program>.

2) Specifications for Feature B012, "Embedded C":

   a) Subclause 21.4, "<embedded SQL C program>":

      i) Without Feature B012, "Embedded C", conforming SQL language shall not contain an <embedded SQL C program>.

3) Specifications for Feature B013, "Embedded COBOL":

   a) Subclause 21.5, "<embedded SQL COBOL program>":

      i) Without Feature B013, "Embedded COBOL", conforming SQL language shall not contain an <embedded SQL COBOL program>.

4) Specifications for Feature B014, "Embedded Fortran":

   a) Subclause 21.6, "<embedded SQL Fortran program>":

      i) Without Feature B014, "Embedded Fortran", conforming SQL language shall not contain an <embedded SQL Fortran program>.

5) Specifications for Feature B015, "Embedded MUMPS":

   a) Subclause 21.7, "<embedded SQL MUMPS program>":

      i) Without Feature B015, "Embedded MUMPS", conforming SQL language shall not contain an <embedded SQL MUMPS program>.

6) Specifications for Feature B016, "Embedded Pascal":

a) Subclause 21.8, “<embedded SQL Pascal program>”:

i) Without Feature B016, “Embedded Pascal”, conforming SQL language shall not contain an <embedded SQL Pascal program>.

7) Specifications for Feature B017, “Embedded PL/I”:

a) Subclause 21.9, “<embedded SQL PL/I program>”:

i) Without Feature B017, “Embedded PL/I”, conforming SQL language shall not contain an <embedded SQL PL/I program>.

8) Specifications for Feature B021, “Direct SQL”:

a) Subclause 22.1, “<direct SQL statement>”:

i) Without Feature B021, “Direct SQL”, conforming SQL language shall not contain a <direct SQL statement>.

9) Specifications for Feature B031, “Basic dynamic SQL”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain an <SQL statement name>.

ii) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain <dynamic cursor name>.

iii) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <descriptor name>.

b) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <general value specification> that contains a <dynamic parameter specification>.

c) Subclause 20.2, “<allocate descriptor statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain an <allocate descriptor statement>.

d) Subclause 20.3, “<deallocate descriptor statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <deallocate descriptor statement>.

e) Subclause 20.4, “<get descriptor statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <get descriptor statement>.

f) Subclause 20.5, “<set descriptor statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <set descriptor statement>.

g) Subclause 20.6, “<prepare statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <prepare statement>.

h) Subclause 20.9, “<describe statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <describe output statement>.

i) Subclause 20.10, “<input using clause>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain an <input using clause>.

j) Subclause 20.11, “<output using clause>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain an <output using clause>.

k) Subclause 20.12, “<execute statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain an <execute statement>.

l) Subclause 20.13, “<execute immediate statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain an <execute immediate statement>.

m) Subclause 20.14, “<dynamic declare cursor>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic declare cursor>.

n) Subclause 20.16, “<dynamic open statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic open statement>.

o) Subclause 20.17, “<dynamic fetch statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic fetch statement>.

p) Subclause 20.18, “<dynamic single row select statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic single row select statement>.

q) Subclause 20.19, “<dynamic close statement>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic close statement>.

r) Subclause 20.20, “<dynamic delete statement: positioned>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic delete statement: positioned>.

s) Subclause 20.21, “<dynamic update statement: positioned>”:

i) Without Feature B031, “Basic dynamic SQL”, conforming SQL language shall not contain a <dynamic update statement: positioned>.

10) Specifications for Feature B032, “Extended dynamic SQL”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <extended statement name> or <extended cursor name>.

ii) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <descriptor name> that is not a <literal>.

b) Subclause 20.2, “<allocate descriptor statement>”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain an <occurrences> that is not a <literal>.

c) Subclause 20.8, “<deallocate prepared statement>”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <deallocate prepared statement>.

d) Subclause 20.9, “<describe statement>”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <describe input statement>.

e) Subclause 20.12, “<execute statement>”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <result using clause>.

f) Subclause 20.15, “<allocate cursor statement>”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain an <allocate cursor statement>.

g) Subclause 20.22, “<preparable dynamic delete statement: positioned>”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <preparable dynamic delete statement: positioned>.

h) Subclause 20.24, “<preparable dynamic update statement: positioned>”:

i) Without Feature B032, “Extended dynamic SQL”, conforming SQL language shall not contain a <preparable dynamic update statement: positioned>.

11) Specifications for Feature B033, “Untyped SQL-invoked function arguments”:

a) Subclause 10.4, “<routine invocation>”:

i) Without Feature B033, “Untyped SQL-invoked function arguments”, conforming SQL language shall not contain a <routine invocation> that is not simply contained in a <call statement> that simply contains an <SQL argument> that is a <dynamic parameter specification>.

12) Specifications for Feature B034, “Dynamic specification of cursor attributes”:

a) Subclause 20.6, “<prepare statement>”:

i) Without Feature B034, “Dynamic specification of cursor attributes”, conforming SQL language shall not contain an <attributes specification>.

13) Specifications for Feature B041, “Extensions to embedded SQL exception declarations”:

a) Subclause 21.2, “<embedded exception declaration>”:

i) Without Feature B041, “Extensions to embedded SQL exception declarations”, conforming SQL language shall not contain an <SQL condition> that contains either SQLSTATE or CONSTRAINT.

14) Specifications for Feature B051, “Enhanced execution rights”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B051, “Enhanced execution rights”, conforming SQL language shall not contain a <module authorization clause> that immediately contains FOR STATIC ONLY or FOR STATIC AND DYNAMIC.

b) Subclause 21.1, “<embedded SQL host program>”:

i) Without Feature B051, “Enhanced execution rights”, conforming SQL language shall not contain an <embedded authorization declaration>.

15) Specifications for Feature B111, “Module language Ada”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B111, “Module language Ada”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains ADA.

16) Specifications for Feature B112, “Module language C”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B112, “Module language C”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains C.

17) Specifications for Feature B113, “Module language COBOL”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B113, “Module language COBOL”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains COBOL.

18) Specifications for Feature B114, “Module language Fortran”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B114, “Module language Fortran”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains FORTRAN.

19) Specifications for Feature B115, “Module language MUMPS”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B115, “Module language MUMPS”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains M.

20) Specifications for Feature B116, “Module language Pascal”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B116, “Module language Pascal”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains PASCAL.

21) Specifications for Feature B117, “Module language PL/I”:

a) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature B117, “Module language PL/I”, conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains PLI.

22) Specifications for Feature B121, “Routine language Ada”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B121, “Routine language Ada”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains ADA.

23) Specifications for Feature B122, “Routine language C”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B122, “Routine language C”, conforming SQL language shall not contain a <routine characteristic>that contains a <language clause> that contains C.

24) Specifications for Feature B123, “Routine language COBOL”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B123, “Routine language COBOL”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains COBOL.

25) Specifications for Feature B124, “Routine language Fortran”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B124, “Routine language Fortran”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains FORTRAN.

26) Specifications for Feature B125, “Routine language MUMPS”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B125, “Routine language MUMPS”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains M.

27) Specifications for Feature B126, “Routine language Pascal”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B126, “Routine language Pascal”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains PASCAL.

28) Specifications for Feature B127, “Routine language PL/I”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B127, “Routine language PL/I”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains PLI.

29) Specifications for Feature B128, “Routine language SQL”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature B128, “Routine language SQL”, conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains SQL.

30) Specifications for Feature F032, “CASCADE drop behavior”:

a) Subclause 11.21, “<drop table statement>”:

i) Without Feature F032, “CASCADE drop behavior”, conforming SQL language shall not contain a <drop table statement> that contains <drop behavior> that contains CASCADE.

b) Subclause 11.23, “<drop view statement>”:

i) Without Feature F032, “CASCADE drop behavior”, conforming SQL language shall not contain a <drop view statement> that contains a <drop behavior> that contains CASCADE.

c) Subclause 11.49, “<drop data type statement>”:

i) Without Feature F032, “CASCADE drop behavior”, conforming SQL language shall not contain a <drop data type statement> that contains a <drop behavior> that contains CASCADE.

d) Subclause 11.52, “<drop routine statement>”:

i) Without Feature F032, “CASCADE drop behavior”, conforming SQL language shall not contain a <drop routine statement> that contains a <drop behavior> that contains CASCADE.

31) Specifications for Feature F033, “ALTER TABLE statement: DROP COLUMN clause”:

a) Subclause 11.18, “<drop column definition>”:

i) Without Feature F033, “ALTER TABLE statement: DROP COLUMN clause”, conforming SQL language shall not contain a <drop column definition>.

32) Specifications for Feature F034, “Extended REVOKE statement”:

a) Subclause 12.7, “<revoke statement>”:

i) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke statement> that contains a <drop behavior> that contains CASCADE.

ii) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke option extension> that contains GRANT OPTION FOR.

iii) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke statement> that contains a <privileges> that contains an <object name> where the owner of the SQL-schema that is specified explicitly or implicitly in the <object name> is not the current authorization identifier.

iv) Without Feature F034, “Extended REVOKE statement”, conforming SQL language shall not contain a <revoke statement> such that there exists a privilege descriptor *PD* that satisfies all the following conditions:

1) *PD* identifies the object identified by <object name> simply contained in <privileges> contained in the <revoke statement>.

2) *PD* identifies the <grantee> identified by any <grantee> simply contained in <revoke statement> and that <grantee> does not identify the owner of the SQL-schema that is specified explicitly or implicitly in the <object name> simply contained in <privileges> contained in the <revoke statement>.

3) *PD* identifies the action identified by the <action> simply contained in <privileges> contained in the <revoke statement>.

4) *PD* indicates that the privilege is grantable.

33) Specifications for Feature F052, "Intervals and datetime arithmetic":

a) Subclause 5.3, "<literal>":

i) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval literal>.

b) Subclause 6.1, "<data type>":

i) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval type>.

c) Subclause 6.27, "<numeric value function>":

i) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <extract expression>.

ii) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <extract expression> that specifies a <time zone field>.

d) Subclause 6.30, "<datetime value expression>":

i) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain <datetime value expression> that immediately contains a <plus sign> or a <minus sign>.

e) Subclause 6.32, "<interval value expression>":

i) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval value expression>.

f) Subclause 6.33, "<interval value function>":

i) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL shall not contain an <interval value function>.

g) Subclause 10.1, "<interval qualifier>":

i) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval qualifier>.

34) Specifications for Feature F053, "OVERLAPS predicate":

a) Subclause 8.14, "<overlaps predicate>":

i) Without Feature F053, "OVERLAPS predicate", conforming SQL language shall not contain an <overlaps predicate>.

35) Specifications for Feature F111, “Isolation levels other than SERIALIZABLE”:

a) Subclause 17.3, “<transaction characteristics>”:

i) Without Feature F111, “Isolation levels other than SERIALIZABLE”, conforming SQL language shall not contain an <isolation level> that contains a <level of isolation> other than SERIALIZABLE.

36) Specifications for Feature F121, “Basic diagnostics management”:

a) Subclause 17.3, “<transaction characteristics>”:

i) Without Feature F121, “Basic diagnostics management”, conforming SQL language shall not contain a <diagnostics size>.

b) Subclause 23.1, “<get diagnostics statement>”:

i) Without Feature F121, “Basic diagnostics management”, conforming SQL language shall not contain a <get diagnostics statement>.

37) Specifications for Feature F122, “Enhanced diagnostics management”:

a) Subclause 23.1, “<get diagnostics statement>”:

i) Without Feature F122, “Enhanced diagnostics management”, conforming SQL language shall not contain a <get diagnostics statement> containing a <simple target specification> whose declared type is different from that of the corresponding <statement information item name> or <condition information item name>.

38) Specifications for Feature F123, “All diagnostics”:

a) Subclause 23.1, “<get diagnostics statement>”:

i) Without Feature F123, “All diagnostics”, conforming SQL language shall not contain an <all information>.

39) Specifications for Feature F171, “Multiple schemas per user”:

a) Subclause 11.1, “<schema definition>”:

i) Without Feature F171, “Multiple schemas per user”, conforming SQL language shall not contain a <schema name clause> that contains a <schema name>.

40) Specifications for Feature F191, “Referential delete actions”:

a) Subclause 11.8, “<referential constraint definition>”:

i) Without Feature F191, “Referential delete actions”, conforming SQL language shall not contain a <delete rule>.

41) Specifications for Feature F222, “INSERT statement: DEFAULT VALUES clause”:

a) Subclause 14.10, “<insert statement>”:

i) Without Feature F222, “INSERT statement: DEFAULT VALUES clause”, conforming SQL language shall not contain a <from default>.

42) Specifications for Feature F251, “Domain support”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <domain name>.

b) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <general value specification> that contains VALUE.

c) Subclause 11.24, “<domain definition>”:

i) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <domain definition>.

d) Subclause 11.30, “<drop domain statement>”:

i) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <drop domain statement>.

43) Specifications for Feature F262, “Extended CASE expression”:

a) Subclause 6.11, “<case expression>”:

i) Without Feature F262, “Extended CASE expression”, in conforming SQL language, a <case operand> immediately contained in a <simple case> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

ii) Without Feature F262, “Extended CASE expression”, in conforming SQL language, a <when operand> contained in a <simple when clause> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

44) Specifications for Feature F263, “Comma-separated predicates in simple CASE expression”:

a) Subclause 6.11, “<case expression>”:

i) Without Feature F263, “Comma-separated predicates in simple CASE expression”, in conforming SQL language, a <when operand list> contained in a <simple when clause> shall simply contain exactly one <when operand>.

45) Specifications for Feature F271, “Compound character literals”:

a) Subclause 5.3, “<literal>”:

i) Without Feature F271, “Compound character literals”, in conforming SQL language, a <character string literal> shall contain exactly one repetition of <character representation> (that is, it shall contain exactly one sequence of “<quote> <character representation>... <quote>”).

46) Specifications for Feature F281, “LIKE enhancements”:

a) Subclause 8.5, “<like predicate>”:

i) Without Feature F281, “LIKE enhancements”, or Feature T022, “Advanced BINARY and VARBINARY data type support”, conforming SQL language shall not contain a <common value expression> simply contained in the <row value predicand> immediately contained in <character like predicate> that is not a column reference.

ii) Without Feature F281, “LIKE enhancements”, conforming SQL language shall not contain a <character pattern> that is not a <value specification>.

iii) Without Feature F281, “LIKE enhancements”, conforming SQL language shall not contain an <escape character> that is not a <value specification>.

47) Specifications for Feature F291, “UNIQUE predicate”:

a) Subclause 8.11, “<unique predicate>”:

i) Without Feature F291, “UNIQUE predicate”, conforming SQL language shall not contain a <unique predicate>.

NOTE 474 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

48) Specifications for Feature F301, “CORRESPONDING in query expressions”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature F301, “CORRESPONDING in query expressions”, conforming SQL language shall not contain a <query expression> that contains CORRESPONDING.

49) Specifications for Feature F302, “INTERSECT table operator”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature F302, “INTERSECT table operator”, conforming SQL language shall not contain a <query term> that contains INTERSECT.

50) Specifications for Feature F304, “EXCEPT ALL table operator”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature F304, “EXCEPT ALL table operator”, conforming SQL language shall not contain a <query expression> that contains EXCEPT ALL.

NOTE 475 — If DISTINCT, INTERSECT or EXCEPT is specified, then the Conformance Rules of Subclause 9.10, “Grouping operations”, apply.

51) Specifications for Feature F312, “MERGE statement”:

a) Subclause 14.11, “<merge statement>”:

i) Without Feature F312, “MERGE statement”, conforming SQL language shall not contain a <merge statement>.

52) Specifications for Feature F321, “User authorization”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature F321, “User authorization”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_USER, SYSTEM_USER, or SESSION_USER.

NOTE 476 — Although CURRENT_USER and USER are semantically the same, without Feature F321, “User authorization”, CURRENT_USER shall be specified as USER.

b) Subclause 11.5, “<default clause>”:

i) Without Feature F321, “User authorization”, conforming SQL language shall not contain a <default option> that contains CURRENT_USER, SESSION_USER, or SYSTEM_USER.

NOTE 477 — Although CURRENT_USER and USER are semantically the same, without Feature F321, “User authorization”, CURRENT_USER shall be specified as USER.

c) Subclause 19.2, “<set session user identifier statement>”:

i) Without Feature F321, “User authorization”, conforming SQL language shall not contain a <set session user identifier statement>.

53) Specifications for Feature F361, “Subprogram support”:

a) Subclause 21.1, “<embedded SQL host program>”:

i) Without Feature F361, “Subprogram support”, conforming SQL language shall not contain two <host variable definition>s that specify the same variable name.

54) Specifications for Feature F381, “Extended schema manipulation”:

a) Subclause 11.2, “<drop schema statement>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop schema statement>.

b) Subclause 11.12, “<alter column definition>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <alter column definition>.

c) Subclause 11.13, “<set column default clause>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <set column default clause>.

d) Subclause 11.14, “<drop column default clause>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop column default clause>.

e) Subclause 11.15, “<add column scope clause>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <add column scope clause>.

f) Subclause 11.16, “<drop column scope clause>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop column scope clause>.

g) Subclause 11.19, “<add table constraint definition>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <add table constraint definition>.

h) Subclause 11.20, “<drop table constraint definition>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop table constraint definition>.

i) Subclause 11.51, “<alter routine statement>”:

i) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <alter routine statement>.

55) Specifications for Feature F391, “Long identifiers”:

a) Subclause 5.2, “<token> and <separator>”:

i) Without Feature F391, “Long identifiers”, in a <regular identifier>, the number of <identifier part>s shall be less than 18.

ii) Without Feature F391, “Long identifiers”, the <delimited identifier body> of a <delimited identifier> shall not comprise more than 18 <delimited identifier part>s.

NOTE 478 — Not every character set supported by a conforming SQL-implementation necessarily contains every character associated with <identifier start> and <identifier part> that is identified in the Syntax Rules of this Subclause. No conforming SQL-implementation shall be required to support in <identifier start> or <identifier part> any character identified in the Syntax Rules of this Subclause unless that character belongs to the character set in use for an SQL-client module or in SQL-data.

56) Specifications for Feature F392, “Unicode escapes in identifiers”:

a) Subclause 5.2, “<token> and <separator>”:

i) Without Feature F392, “Unicode escapes in identifiers”, conforming SQL language shall not contain a <Unicode delimited identifier>.

57) Specifications for Feature F393, “Unicode escapes in literals”:

a) Subclause 5.3, “<literal>”:

i) Without Feature F393, “Unicode escapes in literals”, conforming SQL language shall not contain a <Unicode character string literal>.

58) Specifications for Feature F394, “Optional normal form specification”:

a) Subclause 6.29, “<string value function>”:

i) Without Feature F394, “Optional normal form specification”, conforming SQL language shall not contain <normal form>.

b) Subclause 8.12, “<normalized predicate>”:

i) Without Feature F394, “Optional normal form specification”, conforming SQL language shall not contain <normal form>.

59) Specifications for Feature F401, “Extended joined table”:

a) Subclause 7.7, “<joined table>”:

i) Without Feature F401, “Extended joined table”, conforming SQL language shall not contain a <cross join>.

ii) Without Feature F401, “Extended joined table”, conforming SQL language shall not contain a <natural join>.

iii) Without Feature F401, “Extended joined table”, conforming SQL language shall not contain FULL.

60) Specifications for Feature F402, “Named column joins for LOBs, arrays, and multisets”:

a) Subclause 7.7, “<joined table>”:

i) Without Feature F402, “Named column joins for LOBs, arrays, and multisets”, conforming SQL language shall not contain a <joined table> that simply contains either <natural join> or <named columns join> in which, if *C* is a corresponding join column, the declared type of *C* is LOB-ordered, array-ordered, or multiset-ordered.

NOTE 479 — If *C* is a corresponding join column, then the Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

61) Specifications for Feature F403, “Partitioned join tables”:

a) Subclause 7.7, “<joined table>”:

i) Without Feature F403, “Partitioned join tables”, conforming SQL language shall not contain <partitioned join table>.

62) Specifications for Feature F411, “Time zone specification”:

a) Subclause 5.3, “<literal>”:

i) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <time zone interval>.

b) Subclause 6.1, “<data type>”:

i) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain <with or without time zone>.

c) Subclause 6.27, “<numeric value function>”:

i) Feature F411, “Time zone specification”, conforming SQL language shall not contain an <extract expression> that specifies a <time zone field>.

d) Subclause 6.30, “<datetime value expression>”:

i) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <time zone>.

e) Subclause 6.31, “<datetime value function>”:

i) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <current time value function>.

ii) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <current timestamp value function>.

f) Subclause 19.4, “<set local time zone statement>”:

i) Without Feature F411, “Time zone specification”, conforming SQL language shall not contain a <set local time zone statement>.

63) Specifications for Feature F421, “National character”:

a) Subclause 5.3, “<literal>”:

i) Without Feature F421, “National character”, conforming SQL language shall not contain a <national character string literal>.

b) Subclause 6.1, “<data type>”:

i) Without Feature F421, “National character”, conforming SQL language shall not contain a <national character string type>

c) Subclause 6.12, “<cast specification>”:

i) Without Feature F421, “National character”, conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER LARGE OBJECT.

d) Subclause 6.27, “<numeric value function>”:

i) Without Feature F421, “National character”, conforming SQL language shall not contain a <length expression> that simply contains a <string value expression> that has a declared type of NATIONAL CHARACTER LARGE OBJECT.

e) Subclause 8.5, “<like predicate>”:

i) Without Feature F421, “National character”, and Feature T042, “Extended LOB data type support”, in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.

64) Specifications for Feature F431, “Read-only scrollable cursors”:

a) Subclause 14.2, “<cursor properties>”:

i) Without Feature F431, “Read-only scrollable cursors”, conforming SQL language shall not contain a <cursor scrollability>.

b) Subclause 14.5, “<fetch statement>”:

i) Without Feature F431, “Read-only scrollable cursors”, a <fetch statement> shall not contain a <fetch orientation>.

65) Specifications for Feature F441, “Extended set function support”:

a) Subclause 7.8, “<where clause>”:

i) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <value expression> directly contained in a <where clause> that contains a <column reference> that references a <derived column> that generally contains a <set function specification> without an intervening <routine invocation>.

b) Subclause 10.9, “<aggregate function>”:

i) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <general set function> that contains a <computational operation> that immediately contains COUNT and does not contain a <set quantifier> that immediately contains DISTINCT.

ii) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <general set function> that does not contain a <set quantifier> that immediately contains DISTINCT and that contains a <value expression> that contains a column reference that does not reference a column of *T*.

iii) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <binary set function> that does not contain either a <dependent variable expression> or an <independent variable expression> that contains a column reference that references a column of *T*.

iv) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <value expression> simply contained in a <general set function> that contains a column reference that is an outer reference where the <value expression> is not a column reference.

v) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a <numeric value expression> simply contained in a <dependent variable expression> or an <independent variable expression> that contains a column reference that is an outer reference and in which the <numeric value expression> is not a column reference.

vi) Without Feature F441, “Extended set function support”, conforming SQL language shall not contain a column reference contained in an <aggregate function> that contains a reference to a column derived from a <value expression> that generally contains an <aggregate function> *SFS2* without an intervening <routine invocation>.

66) Specifications for Feature F442, “Mixed column references in set functions”:

a) Subclause 10.9, “<aggregate function>”:

i) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain a <hypothetical set function value expression list> or a <sort specification list> that simply contains a <value expression> that contains more than one column reference, one of which is an outer reference.

ii) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain an <inverse distribution function> that contains an <inverse distribution function argument> or a <sort specification> that contains more than one column reference, one of which is an outer reference.

iii) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain an <aggregate function> that contains a <general set function> whose simply contained <value expression> contains more than one column reference, one of which is an outer reference.

iv) Without Feature F442, “Mixed column references in set functions”, conforming SQL language shall not contain an <aggregate function> that contains a <binary set function> whose simply contained <dependent variable expression> or <independent variable expression> contains more than one column reference, one of which is an outer reference.

67) Specifications for Feature F451, “Character set definition”:

a) Subclause 11.31, “<character set definition>”:

i) Without Feature F451, “Character set definition”, conforming SQL language shall not contain a <character set definition>.

b) Subclause 11.32, “<drop character set statement>”:

i) Without Feature F451, “Character set definition”, conforming SQL language shall not contain a <drop character set statement>.

68) Specifications for Feature F461, “Named character sets”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <character set name>.

b) Subclause 10.5, “<character set specification>”:

i) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <character set specification>.

c) Subclause 11.1, “<schema definition>”:

i) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <schema character set specification>.

d) Subclause 13.2, “<module name clause>”:

i) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <module character set specification>.

e) Subclause 19.7, “<set names statement>”:

i) Without Feature F461, “Named character sets”, conforming SQL language shall not contain a <set names statement>.

f) Subclause 21.1, “<embedded SQL host program>”:

i) Without Feature F461, “Named character sets”, conforming SQL language shall not contain an <embedded character set declaration>.

69) Specifications for Feature F491, “Constraint management”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F491, “Constraint management”, conforming SQL language shall not contain a <constraint name>.

b) Subclause 10.8, “<constraint name definition> and <constraint characteristics>”:

i) Without Feature F491, “Constraint management”, conforming SQL language shall not contain a <constraint name definition>.

c) Subclause 11.29, “<drop domain constraint definition>”:

i) Without Feature F491, “Constraint management”, conforming SQL language shall not contain a <drop domain constraint definition>.

d) Subclause 21.2, “<embedded exception declaration>”:

i) Without Feature F491, “Constraint management”, conforming SQL language shall not contain an <SQL condition> that contains a <constraint name>.

70) Specifications for Feature F521, “Assertions”:

a) Subclause 11.37, “<assertion definition>”:

i) Without Feature F521, “Assertions”, conforming SQL language shall not contain an <assertion definition>.

b) Subclause 11.38, “<drop assertion statement>”:

i) Without Feature F521, “Assertions”, conforming SQL language shall not contain a <drop assertion statement>.

71) Specifications for Feature F531, “Temporary tables”:

a) Subclause 11.3, “<table definition>”:

i) Without Feature F531, “Temporary tables”, conforming SQL language shall not contain a <table scope> and shall not reference any global or local temporary table.

b) Subclause 14.15, “<temporary table declaration>”:

i) Without Feature F531, “Temporary tables”, conforming SQL language shall not contain a <temporary table declaration>.

72) Specifications for Feature F555, “Enhanced seconds precision”:

a) Subclause 5.3, “<literal>”:

i) Without Feature F555, “Enhanced seconds precision”, in conforming SQL language, an <unsigned integer> that is a <seconds fraction> that is contained in a <timestamp literal> shall not contain more than 6 <digit>s.

ii) Without Feature F555, “Enhanced seconds precision”, in conforming SQL language, a <time literal> shall not contain a <seconds fraction>.

b) Subclause 6.1, “<data type>”:

i) Without Feature F555, “Enhanced seconds precision”, conforming SQL language shall not contain a <time precision> that does not specify 0 (zero).

ii) Without Feature F555, “Enhanced seconds precision”, conforming SQL language shall not contain a <timestamp precision> that does not specify either 0 (zero) or 6.

c) Subclause 6.31, “<datetime value function>”:

i) Without Feature F555, “Enhanced seconds precision”, conforming SQL language shall not contain a <current local time value function> that contains a <time precision> that is not 0 (zero).

ii) Without Feature F555, “Enhanced seconds precision”, conforming SQL language shall not contain a <current local timestamp value function> that contains a <timestamp precision> that is neither 0 (zero) nor 6.

73) Specifications for Feature F561, “Full value expressions”:

a) Subclause 8.4, “<in predicate>”:

i) Without Feature F561, “Full value expressions”, conforming SQL language shall not contain a <row value expression> immediately contained in an <in value list> that is not a <value specification>.

NOTE 480 — Since <in predicate> is an equality operation, the Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

b) Subclause 10.9, “<aggregate function>”:

i) Without Feature F561, “Full value expressions”, or Feature F801, “Full set function”, conforming SQL language shall not contain a <general set function> that immediately contains DISTINCT and contains a <value expression> that is not a column reference.

74) Specifications for Feature F571, “Truth value tests”:

a) Subclause 6.34, “<boolean value expression>”:

i) Without Feature F571, “Truth value tests”, conforming SQL language shall not contain a <boolean test> that simply contains a <truth value>.

75) Specifications for Feature F591, “Derived tables”:

a) Subclause 7.6, “<table reference>”:

i) Without Feature F591, “Derived tables”, conforming SQL language shall not contain a <derived table>.

76) Specifications for Feature F611, “Indicator data types”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature F611, “Indicator data types”, in conforming SQL language, the declared types of <indicator parameter>s and <indicator variable>s shall be the same implementation-defined data type.

77) Specifications for Feature F641, “Row and table constructors”:

a) Subclause 7.1, “<row value constructor>”:

i) Without Feature F641, “Row and table constructors”, conforming SQL language shall not contain an <explicit row value constructor> that is not simply contained in a <table value constructor> and that contains more than one <row value constructor element>.

ii) Without Feature F641, “Row and table constructors”, conforming SQL language shall not contain an <explicit row value constructor> that is a <row subquery>.

iii) Without Feature F641, “Row and table constructors”, conforming SQL language shall not contain a <contextually typed row value constructor> that is not simply contained in a <contextually typed table value constructor> and that contains more than one <row value constructor element>.

iv) Without Feature F641, “Row and table constructors”, conforming SQL language shall not contain a <contextually typed row value constructor> that is a <row subquery>.

b) Subclause 7.3, “<table value constructor>”:

i) Without Feature F641, “Row and table constructors”, in conforming SQL language, the <contextually typed row value expression list> of a <contextually typed table value constructor> shall contain exactly one <contextually typed row value constructor> *RVE*. *RVE* shall be of the form “(<contextually typed row value constructor element list>)”.

ii) Without Feature F641, “Row and table constructors”, conforming SQL language shall not contain a <table value constructor>.

78) Specifications for Feature F651, “Catalog name qualifiers”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F651, “Catalog name qualifiers”, conforming SQL language shall not contain a <catalog name>.

b) Subclause 19.5, “<set catalog statement>”:

i) Without Feature F651, “Catalog name qualifiers”, conforming SQL language shall not contain a <set catalog statement>.

79) Specifications for Feature F661, “Simple tables”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature F661, “Simple tables”, conforming SQL language shall not contain a <simple table> that immediately contains a <table value constructor> except in an <insert statement>.

ii) Without Feature F661, “Simple tables”, conforming SQL language shall not contain an <explicit table>.

80) Specifications for Feature F671, “Subqueries in CHECK constraints”:

a) Subclause 11.9, “<check constraint definition>”:

i) Without Feature F671, “Subqueries in CHECK constraints”, conforming SQL language shall not contain a <search condition> contained in a <check constraint definition> that contains a <query expression>.

81) Specifications for Feature F672, “Retrospective check constraints”:

a) Subclause 11.9, “<check constraint definition>”:

i) Without Feature F672, “Retrospective check constraints”, conforming SQL language shall not contain a <check constraint definition> that generally contains CURRENT_DATE, CURRENT_TIMESTAMP, or LOCALTIMESTAMP.

b) Subclause 11.37, “<assertion definition>”:

i) Without Feature F672, “Retrospective check constraints”, conforming SQL language shall not contain an <assertion definition> that generally contains CURRENT_DATE, CURRENT_TIMESTAMP, or LOCALTIMESTAMP.

82) Specifications for Feature F690, “Collation support”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F690, “Collation support”, conforming SQL language shall not contain a <collation name>.

b) Subclause 10.7, “<collate clause>”:

i) Without Feature F690, “Collation support”, conforming SQL language shall not contain a <collate clause>.

c) Subclause 11.33, “<collation definition>”:

i) Without Feature F690, “Collation support”, conforming SQL language shall not contain a <collation definition>.

d) Subclause 11.34, “<drop collation statement>”:

i) Without Feature F690, “Collation support”, conforming SQL language shall not contain a <drop collation statement>.

83) Specifications for Feature F692, “Extended collation support”:

a) Subclause 11.4, “<column definition>”:

i) Without Feature F692, “Extended collation support”, conforming SQL language shall not contain a <column definition> that immediately contains a <collate clause>.

b) Subclause 11.24, “<domain definition>”:

i) Without Feature F692, “Extended collation support”, conforming SQL language shall not contain a <domain definition> that immediately contains a <collate clause>.

c) Subclause 11.42, “<attribute definition>”:

i) Without Feature F692, “Extended collation support”, conforming SQL language shall not contain an <attribute definition> that immediately contains a <collate clause>.

84) Specifications for Feature F693, “SQL-session and client module collations”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature F693, “SQL-session and client module collations”, conforming SQL language shall not contain <current collation specification>.

b) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature F693, “SQL-session and client module collations”, conforming SQL language shall not contain a <module collation specification>.

c) Subclause 19.10, “<set session collation statement>”:

i) Without Feature F693, “SQL-session and client module collations”, conforming SQL language shall not contain a <set session collation statement>.

85) Specifications for Feature F695, “Translation support”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <transliteration name>.

ii) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <transcoding name>.

b) Subclause 6.29, “<string value function>”:

i) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <character transliteration>.

ii) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <transcoding>.

c) Subclause 11.35, “<transliteration definition>”:

i) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <transliteration definition>.

d) Subclause 11.36, “<drop transliteration statement>”:

i) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <drop transliteration statement>.

86) Specifications for Feature F701, “Referential update actions”:

a) Subclause 11.8, “<referential constraint definition>”:

i) Without Feature F701, “Referential update actions”, conforming SQL language shall not contain an <update rule>.

87) Specifications for Feature F711, “ALTER domain”:

a) Subclause 11.25, “<alter domain statement>”:

i) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain an <alter domain statement>.

b) Subclause 11.26, “<set domain default clause>”:

i) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain a <set domain default clause>.

c) Subclause 11.27, “<drop domain default clause>”:

i) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain a <drop domain default clause>.

d) Subclause 11.28, “<add domain constraint definition>”:

i) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain an <add domain constraint definition>.

e) Subclause 11.29, “<drop domain constraint definition>”:

i) Without Feature F711, “ALTER domain”, conforming SQL language shall not contain a <drop domain constraint definition>.

88) Specifications for Feature F721, “Deferrable constraints”:

a) Subclause 10.8, “<constraint name definition> and <constraint characteristics>”:

i) Without Feature F721, “Deferrable constraints”, conforming SQL language shall not contain a <constraint characteristics>.

NOTE 481 — This means that INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

b) Subclause 17.4, “<set constraints mode statement>”:

i) Without Feature F721, “Deferrable constraints”, conforming SQL language shall not contain a <set constraints mode statement>.

89) Specifications for Feature F731, “INSERT column privileges”:

a) Subclause 12.3, “<privileges>”:

i) Without Feature F731, “INSERT column privileges”, in conforming SQL language, an <action> that contains INSERT shall not contain a <privilege column list>.

90) Specifications for Feature F741, “Referential MATCH types”:

a) Subclause 8.13, “<match predicate>”:

i) Without Feature F741, “Referential MATCH types”, conforming SQL language shall not contain a <match predicate>.

NOTE 482 — The Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

b) Subclause 11.8, “<referential constraint definition>”:

i) Without Feature F741, “Referential MATCH types”, conforming SQL language shall not contain a <references specification> that contains MATCH.

91) Specifications for Feature F751, “View CHECK enhancements”:

a) Subclause 11.22, “<view definition>”:

i) Without Feature F751, “View CHECK enhancements”, conforming SQL language shall not contain a <levels clause>.

ii) Without Feature F751, “View CHECK enhancements”, conforming SQL language shall not contain <view definition> that contains a <query expression> that contains a <query expression> and contains WITH CHECK OPTION.

92) Specifications for Feature F761, “Session management”:

a) Subclause 19.1, “<set session characteristics statement>”:

i) Without Feature F761, “Session management”, conforming SQL language shall not contain a <set session characteristics statement>.

b) Subclause 19.5, “<set catalog statement>”:

i) Without Feature F761, “Session management”, conforming SQL language shall not contain a <set catalog statement>.

c) Subclause 19.6, “<set schema statement>”:

i) Without Feature F761, “Session management”, conforming SQL language shall not contain a <set schema statement>.

d) Subclause 19.7, “<set names statement>”:

i) Without Feature F761, “Session management”, conforming SQL language shall not contain a <set names statement>.

93) Specifications for Feature F762, “CURRENT_CATALOG”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature F762, “CURRENT_CATALOG”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_CATALOG.

b) Subclause 11.5, “<default clause>”:

i) Without Feature F762, “CURRENT_CATALOG”, conforming SQL language shall not contain a <default option> that contains CURRENT_CATALOG.

94) Specifications for Feature F763, “CURRENT_SCHEMA”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature F763, “CURRENT_SCHEMA”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_SCHEMA.

b) Subclause 11.5, “<default clause>”:

i) Without Feature F763, “CURRENT_SCHEMA”, conforming SQL language shall not contain a <default option> that contains CURRENT_SCHEMA.

95) Specifications for Feature F771, “Connection management”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F771, “Connection management”, conforming SQL language shall not contain an explicit <connection name>.

b) Subclause 18.1, “<connect statement>”:

i) Without Feature F771, “Connection management”, conforming SQL language shall not contain a <connect statement>.

c) Subclause 18.2, “<set connection statement>”:

i) Without Feature F771, “Connection management”, conforming SQL language shall not contain a <set connection statement>.

d) Subclause 18.3, “<disconnect statement>”:

i) Without Feature F771, “Connection management”, conforming SQL language shall not contain a <disconnect statement>.

96) Specifications for Feature F781, “Self-referencing operations”:

a) Subclause 14.9, “<delete statement: searched>”:

i) Without Feature F781, “Self-referencing operations”, conforming SQL language shall not contain a <delete statement: searched> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in the <search condition>.

b) Subclause 14.10, “<insert statement>”:

i) Without Feature F781, “Self-referencing operations”, conforming SQL language shall not contain an <insert statement> in which the <table name> of a leaf generally underlying table of *T* is generally contained in the <from subquery> except as the table name of a qualifying table of a column reference.

c) Subclause 14.11, “<merge statement>”:

i) Without Feature F781, “Self-referencing operations”, conforming SQL language shall not contain a <merge statement> in which a leaf generally underlying table of *T* is generally contained in a <query expression> immediately contained in the <table reference> except as the <table or query name> or <correlation name> of a column reference.

ii) Without Feature F781, “Self-referencing operations”, conforming SQL language shall not contain a <merge statement> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in the <search condition>.

d) Subclause 14.13, “<update statement: searched>”:

i) Without Feature F781, “Self-referencing operations”, conforming SQL language shall not contain an <update statement: searched> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in the <search condition>.

e) Subclause 14.14, “<set clause list>”:

i) Without Feature F781, “Self-referencing operations”, conforming SQL language shall not contain a <set clause> in which a leaf generally underlying table of *T* is an underlying table of any <query expression> generally contained in any <value expression> simply contained in an <update source> or <assigned row> immediately contained in the <set clause>.

97) Specifications for Feature F791, “Insensitive cursors”:

a) Subclause 14.2, “<cursor properties>”:

i) Without Feature F791, “Insensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains INSENSITIVE.

ii) Without Feature F791, “Insensitive cursors”, or Feature T231, “Sensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains ASENSITIVE.

98) Specifications for Feature F801, “Full set function”:

a) Subclause 7.12, “<query specification>”:

i) Without Feature F801, “Full set function”, conforming SQL language shall not contain a <query specification> *QS* that contains more than one <set quantifier> containing DISTINCT, unless such <set quantifier> is contained in a <query expression> contained in *QS*.

99) Specifications for Feature F821, “Local table references”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature F821, “Local table references”, conforming SQL language shall not contain a <local or schema qualifier> that contains a <local qualifier>.

b) Subclause 6.7, “<column reference>”:

i) Without Feature F821, “Local table references”, conforming SQL language shall not contain a <column reference> that simply contains MODULE.

100) Specifications for Feature F831, “Full cursor update”:

a) Subclause 14.3, “<cursor specification>”:

i) Without Feature F831, “Full cursor update”, conforming SQL language shall not contain an <updatability clause> that contains FOR UPDATE and that contains a <cursor scrollability>.

ii) Without Feature F831, “Full cursor update”, conforming SQL language shall not contain an <updatability clause> that specifies FOR UPDATE and that contains an <order by clause>.

b) Subclause 14.12, “<update statement: positioned>”:

i) Without Feature F831, “Full cursor update”, conforming SQL language shall not contain an <update statement: positioned> in which the declared <cursor specification> of *CDD* is ordered.

101) Specifications for Feature F841, “LIKE_REGEX predicate”:

a) Subclause 8.7, “<regex like predicate>”:

i) Without Feature F841, “LIKE_REGEX predicate”, conforming SQL language shall not contain <regex like predicate>.

102) Specifications for Feature F842, “OCCURENCES_REGEX function”:

a) Subclause 6.27, “<numeric value function>”:

i) Without Feature F842, “OCCURENCES_REGEX function”, conforming SQL language shall not contain <regex occurrences function>.

103) Specifications for Feature F843, “POSITION_REGEX function”:

a) Subclause 6.27, “<numeric value function>”:

i) Without Feature F843, “POSITION_REGEX function”, conforming SQL language shall not contain <regex position expression>.

104) Specifications for Feature F844, “SUBSTRING_REGEXP”:

a) Subclause 6.29, “<string value function>”:

i) Without Feature F844, “SUBSTRING_REGEXP”, conforming SQL language shall not contain <regex substring function>.

105) Specifications for Feature F845, “TRANSLATE_REGEX”:

a) Subclause 6.29, “<string value function>”:

i) Without Feature F845, “TRANSLATE_REGEX”, conforming SQL language shall not contain <regex transliteration>.

106) Specifications for Feature F846, “Octet support in regular expression operators”:

a) Subclause 6.27, “<numeric value function>”:

i) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language, <regex occurrences function> shall not contain <char length units>.

ii) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language , <regex position expression> shall not contain <char length units>.

b) Subclause 6.29, “<string value function>”:

i) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language, <regex substring function> shall not contain <char length units>.

ii) Without Feature F846, “Octet support in regular expression operators”, in conforming SQL language, <regex transliteration> shall not contain <char length units>.

107) Specifications for Feature F847, “Nonconstant regular expression”:

a) Subclause 6.27, “<numeric value function>”:

i) Without Feature F847, “Nonconstant regular expression”, in conforming SQL language, <XQuery pattern> and <XQuery option flag> shall be <value specification>s.

b) Subclause 6.29, “<string value function>”:

i) Without Feature F847, “Nonconstant regular expressions”, in conforming SQL language, <XQuery pattern>, <XQuery option flag>, and <XQuery replacement string> shall be <value specification>s.

c) Subclause 8.7, “<regex like predicate>”:

i) Without Feature F847, “Nonconstant regular expressions”, in conforming SQL language, <XQuery pattern> and <XQuery option flag> shall be <value specification>s.

108) Specifications for Feature S023, “Basic structured types”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <attribute name>.

b) Subclause 6.1, “<data type>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <path-resolved user-defined type name> that identifies a structured type.

c) Subclause 6.16, “<method invocation>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <method invocation>.

d) Subclause 6.18, “<new specification>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <new specification>.

e) Subclause 10.4, “<routine invocation>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <generalized expression>.

f) Subclause 11.41, “<user-defined type definition>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <member list>.

ii) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <method specification list>.

g) Subclause 11.42, “<attribute definition>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain an <attribute definition>.

h) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <method specification designator>.

i) Subclause 12.3, “<privileges>”:

i) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <privileges> that contains an <action> that contains UNDER and that contains an <object name> that contains a <schema-resolved user-defined type name> that identifies a structured type.

109) Specifications for Feature S024, “Enhanced structured types”:

a) Subclause 6.17, “<static method invocation>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <static method invocation>.

b) Subclause 9.9, “Equality operations”:

i) Without Feature S024, “Enhanced structured types”, in conforming SQL language, the declared type of an operand of an equality operation shall not be ST-ordered.

c) Subclause 9.10, “Grouping operations”:

i) Without Feature S024, “Enhanced structured types”, in conforming SQL language, the declared type of an operand of a grouping operation shall not be ST-ordered.

d) Subclause 9.11, “Multiset element grouping operations”:

i) Without Feature S024, “Enhanced structured types”, in conforming SQL language, the declared element type of a multiset operand of a multiset element grouping operation shall not be ST-ordered.

e) Subclause 9.12, “Ordering operations”:

i) Without Feature S024, “Enhanced structured types”, in conforming SQL language, the declared type of an operand of an ordering operation shall not be ST-ordered.

f) Subclause 10.6, “<specific routine designator>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <specific routine designator> that contains a <routine type> that immediately contains METHOD.

g) Subclause 11.41, “<user-defined type definition>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain an <instantiable clause> that contains NOT INSTANTIABLE.

ii) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain an <original method specification> that immediately contains SELF AS RESULT.

iii) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <method characteristics> that contains a <parameter style> that contains GENERAL.

iv) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain an <original method specification> that contains an <SQL-data access indication> that immediately contains NO SQL.

v) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <partial method specification> that contains INSTANCE or STATIC.

h) Subclause 11.42, “<attribute definition>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain an <attribute default>.

i) Subclause 11.43, “<alter type statement>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain an <alter type statement>.

j) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S024, “Enhanced structured types”, an <SQL parameter declaration> shall not contain RESULT.

k) Subclause 11.52, “<drop routine statement>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <drop routine statement> that contains a <specific routine designator> that identifies a method.

l) Subclause 12.2, “<grant privilege statement>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <specific routine designator> contained in a <grant privilege statement> that identifies a method.

m) Subclause 12.3, “<privileges>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <privileges> that contains an <action> that contains USAGE and that contains an <object name> that contains a <schema-resolved user-defined type name> that identifies a structured type.

ii) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <privilege method list>.

n) Subclause 14.10, “<insert statement>”:

i) Without Feature S024, “Enhanced structured types”, in conforming SQL language, for each column *C* identified in the explicit or implicit <insert column list>, if the declared type of *C* is a structured type *TY*, then the declared type of the corresponding column of the <query expression> or <contextually typed table value constructor> shall be *TY*.

o) Subclause 14.11, “<merge statement>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <merge statement> that does not satisfy the condition: for each column *C* identified in the explicit or implicit <insert column list>, if the declared type of *C* is a structured type *TY*, then the declared type of the corresponding column of the <query expression> or <contextually typed table value constructor> is *TY*.

p) Subclause 14.14, “<set clause list>”:

i) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <set clause> in which the declared type of the <update target> in the <set clause> is a structured type *TY* and the declared type of the <update source> or corresponding field of the <assigned row> contained in the <set clause> is not *TY*.

ii) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <set clause> that contains a <mutated set clause> and in which the declared type of the last <method name> identifies a structured type *TY*, and the declared type of the <update source> contained in the <set clause> is not *TY*.

110) Specifications for Feature S025, “Final structured types”:

a) Subclause 11.41, “<user-defined type definition>”:

i) Without Feature S025, “Final structured types”, in conforming SQL language, a <user-defined type definition> that defines a structured type shall contain a <finality> that is NOT FINAL.

111) Specifications for Feature S026, “Self-referencing structured types”:

a) Subclause 11.42, “<attribute definition>”:

i) Without Feature S026, “Self-referencing structured types”, conforming SQL language shall not contain a <data type> simply contained in an <attribute definition> that is a <reference type> whose <referenced type> is equivalent to the <schema-resolved user-defined type name> simply contained in the <user-defined type definition> that contains <attribute definition>.

112) Specifications for Feature S027, “Create method by specific method name”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S027, “Create method by specific method name”, conforming SQL language shall not contain a <method specification designator> that contains SPECIFIC METHOD.

113) Specifications for Feature S028, “Permutable UDT options list”:

a) Subclause 11.41, “<user-defined type definition>”:

i) Without Feature S028, “Permutable UDT options list”, conforming SQL language shall not contain a <user-defined type option list> in which <instantiable clause>, if specified, <finality>, <reference type specification>, if specified, <cast to ref>, if specified, <cast to type>, if specified, <cast to distinct>, if specified, and <cast to source>, if specified, do not appear in that sequence.

114) Specifications for Feature S041, “Basic reference types”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <reference type>.

b) Subclause 6.19, “<attribute or method reference>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain an <attribute or method reference>.

c) Subclause 6.20, “<dereference operation>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <dereference operation>.

d) Subclause 6.25, “<value expression>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <reference value expression>.

e) Subclause 21.3, “<embedded SQL Ada program>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain an <Ada REF variable>.

f) Subclause 21.4, “<embedded SQL C program>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <C REF variable>.

g) Subclause 21.5, “<embedded SQL COBOL program>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <COBOL REF variable>.

h) Subclause 21.6, “<embedded SQL Fortran program>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <Fortran REF variable>.

i) Subclause 21.7, “<embedded SQL MUMPS program>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <MUMPS REF variable>.

j) Subclause 21.8, “<embedded SQL Pascal program>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <Pascal REF variable>.

k) Subclause 21.9, “<embedded SQL PL/I program>”:

i) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain a <PL/I REF variable>.

115) Specifications for Feature S043, “Enhanced reference types”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <scope clause> that is not simply contained in a <data type> that is simply contained in a <column definition>.

b) Subclause 6.12, “<cast specification>”:

i) Without Feature S043, “Enhanced reference types”, in conforming SQL language, if the declared data type of <cast operand> is a reference type, then <cast target> shall contain a <data type> that is a reference type.

c) Subclause 6.21, “<method reference>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <method reference>.

d) Subclause 6.22, “<reference resolution>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <reference resolution>.

e) Subclause 11.3, “<table definition>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <column option list> that contains a <scope clause>.

ii) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain <reference generation> that does not contain SYSTEM GENERATED.

f) Subclause 11.15, “<add column scope clause>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain an <add column scope clause>.

g) Subclause 11.16, “<drop column scope clause>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <drop column scope clause>.

h) Subclause 11.22, “<view definition>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <referenceable view specification>.

i) Subclause 11.41, “<user-defined type definition>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <reference type specification>.

j) Subclause 14.10, “<insert statement>”:

i) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain an <override clause>.

116) Specifications for Feature S051, “Create table of type”:

a) Subclause 11.3, “<table definition>”:

i) Without Feature S051, “Create table of type”, conforming SQL language shall not contain “OF <path-resolved user-defined type name>”.

117) Specifications for Feature S071, “SQL paths in function and type name resolution”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <general value specification> that contains CURRENT_PATH.

b) Subclause 10.3, “<path specification>”:

i) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <path specification>.

c) Subclause 11.1, “<schema definition>”:

i) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <schema path specification>.

d) Subclause 11.5, “<default clause>”:

i) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <default option> that contains CURRENT_PATH.

e) Subclause 13.1, “<SQL-client module definition>”:

i) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain a <module path specification>.

f) Subclause 19.8, “<set path statement>”:

i) Without Feature S071, “SQL paths in function and type name resolution”, Conforming SQL language shall not contain a <set path statement>.

g) Subclause 21.1, “<embedded SQL host program>”:

i) Without Feature S071, “SQL paths in function and type name resolution”, conforming SQL language shall not contain an <embedded path specification>.

118) Specifications for Feature S081, “Subtables”:

a) Subclause 11.3, “<table definition>”:

i) Without Feature S081, “Subtables”, conforming SQL language shall not contain a <subtable clause>.

b) Subclause 12.2, “<grant privilege statement>”:

i) Without Feature S081, “Subtables”, conforming SQL language shall not contain a <grant privilege statement> that contains WITH HIERARCHY OPTION.

c) Subclause 12.3, “<privileges>”:

i) Without Feature S081, “Subtables”, conforming SQL language shall not contain a <privileges> that contains an <action> that contains UNDER and that contains an <object name> that contains a <table name>.

d) Subclause 12.7, “<revoke statement>”:

i) Without Feature S081, “Subtables”, conforming SQL language shall not contain a <revoke option extension> that contains HIERARCHY OPTION FOR.

119) Specifications for Feature S091, “Basic array support”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array type>.

b) Subclause 6.5, “<contextually typed value specification>”:

i) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <empty specification> that simply contains ARRAY.

c) Subclause 6.23, “<array element reference>”:

i) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array element reference>.

d) Subclause 6.27, “<numeric value function>”:

i) Without Feature S091, “Basic array support”, or Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <cardinality expression>.

e) Subclause 6.35, “<array value expression>”:

i) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array value expression>.

f) Subclause 6.36, “<array value constructor>”:

i) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <array value constructor by enumeration>.

g) Subclause 7.6, “<table reference>”:

i) Without Feature S091, “Basic array support”, or Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <collection derived table>.

h) Subclause 14.14, “<set clause list>”:

i) Without Feature S091, “Basic array support”, conforming SQL language shall not contain an <update target> that immediately contains a <simple value specification>.

120) Specifications for Feature S092, “Arrays of user-defined types”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S092, “Arrays of user-defined types”, conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

121) Specifications for Feature S094, “Arrays of reference types”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S094, “Arrays of reference types”, conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <reference type>.

122) Specifications for Feature S095, “Array constructors by query”:

a) Subclause 6.36, “<array value constructor>”:

i) Without Feature S095, “Array constructors by query”, conforming SQL language shall not contain an <array value constructor by query>.

123) Specifications for Feature S096, “Optional array bounds”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S096, “Optional array bounds”, conforming SQL language shall not contain an <array type> that does not immediately contain <maximum cardinality>.

124) Specifications for Feature S097, “Array element assignment”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature S097, “Array element assignment”, conforming SQL language shall not contain a <target array element specification>.

125) Specifications for Feature S111, “ONLY in query expressions”:

a) Subclause 7.6, “<table reference>”:

i) Without Feature S111, “ONLY in query expressions”, conforming SQL language shall not contain a <table reference> that contains an <only spec>.

b) Subclause 14.8, “<delete statement: positioned>”:

i) Without Feature S111, “ONLY in query expressions”, conforming SQL language shall not contain a <target table> that contains ONLY.

126) Specifications for Feature S151, “Type predicate”:

a) Subclause 8.19, “<type predicate>”:

i) Without Feature S151, “Type predicate”, conforming SQL language shall not contain a <type predicate>.

127) Specifications for Feature S161, “Subtype treatment”:

a) Subclause 6.15, “<subtype treatment>”:

i) Without Feature S161, “Subtype treatment”, conforming SQL Language shall not contain a <subtype treatment>.

128) Specifications for Feature S162, “Subtype treatment for references”:

a) Subclause 6.15, “<subtype treatment>”:

i) Without Feature S162, “Subtype treatment for references”, conforming SQL language shall not contain a <target subtype> that contains a <reference type>.

129) Specifications for Feature S201, “SQL-invoked routines on arrays”:

a) Subclause 10.4, “<routine invocation>”:

i) Without Feature S201, “SQL-invoked routines on arrays”, conforming SQL language shall not contain an <SQL argument> whose declared type is an array type.

b) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S201, “SQL-invoked routines on arrays”, conforming SQL language shall not contain a <parameter type> that is based on an array type.

ii) Without Feature S201, “SQL-invoked routines on arrays”, conforming SQL language shall not contain a <returns data type> that is based on an array type.

130) Specifications for Feature S202, “SQL-invoked routines on multisets”:

a) Subclause 10.4, “<routine invocation>”:

i) Without Feature S202, “SQL-invoked routines on multisets”, conforming SQL language shall not contain an <SQL argument> whose declared type is a multiset type.

b) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S202, “SQL-invoked routines on multisets”, conforming SQL language shall not contain a <parameter type> that is based on a multiset type.

ii) Without Feature S202, “SQL-invoked routines on multisets”, conforming SQL language shall not contain a <returns data type> that is based on a multiset type.

131) Specifications for Feature S211, “User-defined cast functions”:

a) Subclause 11.53, “<user-defined cast definition>”:

i) Without Feature S211, “User-defined cast functions”, conforming SQL language shall not contain a <user-defined cast definition>.

b) Subclause 11.54, “<drop user-defined cast statement>”:

i) Without Feature S211, “User-defined cast functions”, conforming SQL language shall not contain a <drop user-defined cast statement>.

132) Specifications for Feature S231, “Structured type locators”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S231, “Structured type locators”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a structured type.

ii) Without Feature S231, “Structured type locators”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a structured type.

b) Subclause 13.3, “<externally-invoked procedure>”:

i) Without Feature S231, “Structured type locators”, conforming SQL language shall not contain a <host parameter data type> that simply contains a <data type> that specifies a structured type and that contains a <locator indication>.

c) Subclause 21.3, “<embedded SQL Ada program>”:

i) 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.

d) Subclause 21.4, “<embedded SQL C program>”:

i) 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.

e) Subclause 21.5, “<embedded SQL COBOL program>”:

i) 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.

f) Subclause 21.6, “<embedded SQL Fortran program>”:

i) 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.

g) Subclause 21.7, “<embedded SQL MUMPS program>”:

i) 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.

h) Subclause 21.8, “<embedded SQL Pascal program>”:

i) 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.

i) Subclause 21.9, “<embedded SQL PL/I program>”:

i) 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.

133) Specifications for Feature S232, “Array locators”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies an array type.

ii) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies an array type.

b) Subclause 13.3, “<externally-invoked procedure>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <host parameter data type> that simply contains an <array type> and that contains a <locator indication>.

c) Subclause 21.3, “<embedded SQL Ada program>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain an <Ada array locator variable>.

d) Subclause 21.4, “<embedded SQL C program>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain an <C array locator variable>.

e) Subclause 21.5, “<embedded SQL COBOL program>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <COBOL array locator variable>.

f) Subclause 21.6, “<embedded SQL Fortran program>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <Fortran array locator variable>.

g) Subclause 21.7, “<embedded SQL MUMPS program>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <MUMPS array locator variable>.

h) Subclause 21.8, “<embedded SQL Pascal program>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <Pascal array locator variable>.

i) Subclause 21.9, “<embedded SQL PL/I program>”:

i) Without Feature S232, “Array locators”, conforming SQL language shall not contain a <PL/I array locator variable>.

134) Specifications for Feature S233, “Multiset locators”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a multiset type.

ii) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a multiset type.

b) Subclause 13.3, “<externally-invoked procedure>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <host parameter data type> that simply contains a <multiset type> and that contains a <locator indication>.

c) Subclause 21.3, “<embedded SQL Ada program>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain an <Ada multiset locator variable>.

d) Subclause 21.4, “<embedded SQL C program>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <C multiset locator variable>.

e) Subclause 21.5, “<embedded SQL COBOL program>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <COBOL multiset locator variable>.

f) Subclause 21.6, “<embedded SQL Fortran program>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <Fortran multiset locator variable>.

g) Subclause 21.7, “<embedded SQL MUMPS program>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <MUMPS multiset locator variable>.

h) Subclause 21.8, “<embedded SQL Pascal program>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <Pascal multiset locator variable>.

i) Subclause 21.9, “<embedded SQL PL/I program>”:

i) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <PL/I multiset locator variable>.

135) Specifications for Feature S241, “Transform functions”:

a) Subclause 6.4, “<value specification> and <target specification>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain CURRENT_DEFAULT_TRANSFORM_GROUP.

   ii) Without Feature S241, “Transform functions”, conforming SQL language shall not contain CURRENT_TRANSFORM_GROUP_FOR_TYPE.

b) Subclause 11.50, “<SQL-invoked routine>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <transform group specification>.

c) Subclause 11.57, “<transform definition>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <transform definition>.

d) Subclause 11.61, “<drop transform statement>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <drop transform statement>.

e) Subclause 13.1, “<SQL-client module definition>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <module transform group specification>.

f) Subclause 19.9, “<set transform group statement>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <set transform group statement>.

g) Subclause 21.1, “<embedded SQL host program>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <embedded transform group specification>.

h) Subclause 21.3, “<embedded SQL Ada program>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain an <Ada user-defined type variable>.

i) Subclause 21.4, “<embedded SQL C program>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <C user-defined type variable>.

j) Subclause 21.5, “<embedded SQL COBOL program>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <COBOL user-defined type variable>.

k) Subclause 21.6, “<embedded SQL Fortran program>”:

   i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <Fortran user-defined type variable>.

l) Subclause 21.7, “<embedded SQL MUMPS program>”:

i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <MUMPS user-defined type variable>.

m) Subclause 21.8, “<embedded SQL Pascal program>”:

i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <Pascal user-defined type variable>.

n) Subclause 21.9, “<embedded SQL PL/I program>”:

i) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <PL/I user-defined type variable>.

136) Specifications for Feature S242, “Alter transform statement”:

a) Subclause 11.58, “<alter transform statement>”:

i) Without Feature S242, “Alter transform statement”, conforming SQL language shall not contain an <alter transform statement>.

137) Specifications for Feature S251, “User-defined orderings”:

a) Subclause 11.55, “<user-defined ordering definition>”:

i) Without Feature S251, “User-defined orderings”, conforming SQL shall not contain a <user-defined ordering definition>.

NOTE 483 — If MAP is specified, then the Conformance Rules of Subclause 9.9, “Equality operations”, apply. If ORDER FULL BY MAP is specified, then the Conformance Rules of Subclause 9.12, “Ordering operations”, also apply.

b) Subclause 11.56, “<drop user-defined ordering statement>”:

i) Without Feature S251, “User-defined orderings”, conforming SQL language shall not contain a <drop user-defined ordering statement>.

138) Specifications for Feature S261, “Specific type method”:

a) Subclause 6.29, “<string value function>”:

i) Without Feature S261, “Specific type method”, conforming SQL language shall not contain a <specific type method>.

139) Specifications for Feature S271, “Basic multiset support”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <multiset type>.

b) Subclause 6.5, “<contextually typed value specification>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain an <empty specification> that simply contains MULTISET.

c) Subclause 6.24, “<multiset element reference>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <multiset element reference>.

d) Subclause 6.27, “<numeric value function>”:

i) Without Feature S091, “Basic array support”, or Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <cardinality expression>.

e) Subclause 6.38, “<multiset value function>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <multiset value function>.

NOTE 484 — The Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

f) Subclause 6.39, “<multiset value constructor>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <multiset value constructor>.

g) Subclause 7.6, “<table reference>”:

i) Without Feature S091, “Basic array support”, or Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <collection derived table>.

h) Subclause 8.16, “<member predicate>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <member predicate>.

NOTE 485 — The Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

i) Subclause 8.18, “<set predicate>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <set predicate>.

NOTE 486 — The Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

j) Subclause 10.9, “<aggregate function>”:

i) Without Feature S271, “Basic multiset support”, conforming SQL language shall not contain a <computational operation> that immediately contains COLLECT.

140) Specifications for Feature S272, “Multisets of user-defined types”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S272, “Multisets of user-defined types”, conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

141) Specifications for Feature S274, “Multisets of reference types”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S274, “Multisets of reference types”, conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <reference type>.

142) Specifications for Feature S275, “Advanced multiset support”:

a) Subclause 6.37, “<multiset value expression>”:

i) Without Feature S275, “Advanced multiset support”, conforming SQL language shall not contain MULTISET UNION, MULTISET INTERSECTION, or MULTISET EXCEPT.

NOTE 487 — If MULTISET UNION DISTINCT, MULTISET INTERSECTION, or MULTISET EXCEPT is specified, then the Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

b) Subclause 8.17, “<submultiset predicate>”:

i) Without Feature S275, “Advanced multiset support”, conforming SQL language shall not contain a <submultiset predicate>.

NOTE 488 — The Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

c) Subclause 9.9, “Equality operations”:

i) Without Feature S275, “Advanced multiset support”, in conforming SQL language, the declared type of an operand of an equality operation shall not be multiset-ordered.

NOTE 489 — If the declared type of an operand *OP* of an equality operation is a multiset type, then *OP* is a multiset operand of a multiset element grouping operation. The Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, apply.

d) Subclause 10.9, “<aggregate function>”:

i) Without Feature S275, “Advanced multiset support”, conforming SQL language shall not contain a <computational operation> that immediately contains FUSION or INTERSECTION.

NOTE 490 — If INTERSECTION is specified, then the Conformance Rules of Subclause 9.11, “Multiset element grouping operations”, also apply.

143) Specifications for Feature S281, “Nested collection types”:

a) Subclause 6.1, “<data type>”:

i) Without Feature S281, “Nested collection types”, conforming SQL language shall not contain a collection type that is based on a <data type> that contains a <collection type>.

144) Specifications for Feature S291, “Unique constraint on entire row”:

a) Subclause 11.7, “<unique constraint definition>”:

i) Without Feature S291, “Unique constraint on entire row”, conforming SQL language shall not contain UNIQUE(VALUE).

145) Specifications for Feature T021, “BINARY and VARBINARY data types”:

a) Subclause 5.3, “<literal>”:

i) Without Feature T041, “Basic LOB data type support”, or Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <binary string literal>.

b) Subclause 6.1, “<data type>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <binary string type> that is not a <binary large object string type>.

c) Subclause 21.3, “<embedded SQL Ada program>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain an <Ada BINARY variable>.

ii) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain an <Ada VARBINARY variable>.

d) Subclause 21.4, “<embedded SQL C program>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <C BINARY variable>.

ii) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <C VARBINARY variable>.

e) Subclause 21.5, “<embedded SQL COBOL program>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <COBOL BINARY variable>.

f) Subclause 21.6, “<embedded SQL Fortran program>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <Fortran BINARY variable>.

ii) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <Fortran VARBINARY variable>.

g) Subclause 21.7, “<embedded SQL MUMPS program>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <MUMPS VARBINARY variable>.

h) Subclause 21.8, “<embedded SQL Pascal program>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <Pascal BINARY variable>.

i) Subclause 21.9, “<embedded SQL PL/I program>”:

i) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <PL/I BINARY variable>.

ii) Without Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <PL/I VARBINARY variable>.

146) Specifications for Feature T022, “Advanced BINARY and VARBINARY data type support”:

a) Subclause 6.29, “<string value function>”:

i) Without Feature T042, “Extended LOB data type support”, or Feature T022, “Advanced BINARY and VARBINARY data type support”, conforming SQL language shall not contain a <binary value function>.

b) Subclause 8.5, “<like predicate>”:

i) Without Feature F281, “LIKE enhancements”, or Feature T022, “Advanced BINARY and VARBINARY data type support”, conforming SQL language shall not contain a <common value expression> simply contained in the <row value predicand> immediately contained in <character like predicate> that is not a column reference.

147) Specifications for Feature T023, “Compound binary literals”:

a) Subclause 5.3, “<literal>”:

i) Without Feature T023, “Compound binary literals”, in conforming SQL language, a <binary string literal> shall contain exactly one repetition of “<quote> [{ <hexit> <hexit> }...] <quote>”.

148) Specifications for Feature T024, “Spaces in binary literals”:

a) Subclause 5.3, “<literal>”:

i) Without Feature T024, “Spaces in binary literals”, in conforming SQL language, a <binary string literal> shall not contain a <space>.

149) Specifications for Feature T031, “BOOLEAN data type”:

a) Subclause 5.3, “<literal>”:

i) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <boolean literal>.

b) Subclause 6.1, “<data type>”:

i) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <boolean type>.

c) Subclause 6.25, “<value expression>”:

i) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <value expression> that is a <boolean value expression>.

d) Subclause 6.34, “<boolean value expression>”:

i) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <boolean primary> that simply contains a <nonparenthesized value expression primary>.

e) Subclause 7.1, “<row value constructor>”:

i) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <row value constructor predicand> that immediately contains a <boolean predicand>.

f) Subclause 10.9, “<aggregate function>”:

i) Without Feature T031, “BOOLEAN data type”, conforming SQL language shall not contain a <computational operation> that immediately contains EVERY, ANY, or SOME.

150) Specifications for Feature T041, “Basic LOB data type support”:

a) Subclause 5.3, “<literal>”:

i) Without Feature T041, “Basic LOB data type support”, or Feature T021, “BINARY and VARBINARY data types”, conforming SQL language shall not contain a <binary string literal>.

b) Subclause 6.1, “<data type>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <binary large object string type>, a <character large object type>, or a <national character large object type>.

c) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a large object type.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a large object type.

d) Subclause 21.3, “<embedded SQL Ada program>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain an <Ada BLOB variable>.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain an <Ada CLOB variable>.

iii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain an <Ada BLOB locator variable>.

iv) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain an <Ada CLOB locator variable>.

e) Subclause 21.4, “<embedded SQL C program>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <C BLOB variable>.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <C CLOB variable>.

iii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <C BLOB locator variable>.

iv) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <C CLOB locator variable>.

f) Subclause 21.5, “<embedded SQL COBOL program>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <COBOL BLOB variable>.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <COBOL CLOB variable>.

iii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <COBOL BLOB locator variable>.

iv) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <COBOL CLOB locator variable>.

g) Subclause 21.6, “<embedded SQL Fortran program>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <Fortran BLOB variable>.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <Fortran CLOB variable>.

iii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <Fortran BLOB locator variable>.

iv) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <Fortran CLOB locator variable>.

h) Subclause 21.7, “<embedded SQL MUMPS program>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <MUMPS BLOB variable>.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <MUMPS CLOB variable>.

iii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <MUMPS BLOB locator variable>.

iv) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a and <MUMPS CLOB locator variable>.

i) Subclause 21.8, “<embedded SQL Pascal program>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <Pascal BLOB variable>.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <Pascal CLOB variable>.

iii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <Pascal BLOB locator variable>.

iv) 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>.

j) Subclause 21.9, “<embedded SQL PL/I program>”:

i) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <PL/I BLOB variable>.

ii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <PL/I CLOB variable>.

iii) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <PL/I BLOB locator variable>.

iv) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not contain a <PL/I CLOB locator variable>.

151) Specifications for Feature T042, “Extended LOB data type support”:

a) Subclause 6.12, “<cast specification>”:

i) Without Feature T042, “Extended LOB data type support”, conforming SQL language shall not contain a <cast operand> whose declared type is BINARY LARGE OBJECT or CHARACTER LARGE OBJECT.

ii) Without Feature T042, “Extended LOB data type support”, conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER LARGE OBJECT.

b) Subclause 6.29, “<string value function>”:

i) Without Feature T042, “Extended LOB data type support”, or Feature T022, “Advanced BINARY and VARBINARY data type support”, conforming SQL language shall not contain a <binary value function>.

c) Subclause 8.5, “<like predicate>”:

i) Without Feature T042, “Extended LOB data type support”, conforming SQL language shall not contain an <octet like predicate>.

ii) Without Feature T042, “Extended LOB data type support”, in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type CHARACTER LARGE OBJECT

iii) Without Feature F421, “National character”, and Feature T042, “Extended LOB data type support”, in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.

d) Subclause 8.6, “<similar predicate>”:

i) Without Feature T042, “Extended LOB data type support”, in conforming SQL language, a <character value expression> simply contained in a <similar predicate> shall not be of declared type CHARACTER LARGE OBJECT.

e) Subclause 9.9, “Equality operations”:

i) Without Feature T042, “Extended LOB data type support”, in conforming SQL language, the declared type of an operand of an equality operation shall not be LOB-ordered.

152) Specifications for Feature T043, “Multiplier T”:

a) Subclause 5.2, “<token> and <separator>”:

i) Without Feature T043, “Multiplier T”, in conforming SQL language, a <multiplier> shall not be T.

153) Specifications for Feature T044, “Multiplier P”:

a) Subclause 5.2, “<token> and <separator>”:

i) Without Feature T044, “Multiplier P”, in conforming SQL language, a <multiplier> shall not be P.

154) Specifications for Feature T051, “Row types”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature T051, “Row types”, conforming SQL language shall not contain a <field name>.

b) Subclause 6.1, “<data type>”:

i) Without Feature T051, “Row types”, conforming SQL language shall not contain a <row type>.

c) Subclause 6.2, “<field definition>”:

i) Without Feature T051, “Row types”, conforming SQL language shall not contain a <field definition>.

d) Subclause 6.14, “<field reference>”:

i) Without Feature T051, “Row types”, conforming SQL language shall not contain a <field reference>.

e) Subclause 7.1, “<row value constructor>”:

i) Without Feature T051, “Row types”, conforming SQL language shall not contain an <explicit row value constructor> that immediately contains ROW.

ii) Without Feature T051, “Row types”, conforming SQL language shall not contain a <contextually typed row value constructor> that immediately contains ROW.

f) Subclause 7.2, “<row value expression>”:

i) Without Feature T051, “Row types”, conforming SQL language shall not contain a <row value special case>.

g) Subclause 7.12, “<query specification>”:

i) Without Feature T051, “Row types”, conforming SQL language shall not contain an <all fields reference>.

155) Specifications for Feature T052, “MAX and MIN for row types”:

a) Subclause 10.9, “<aggregate function>”:

i) Without Feature T052, “MAX and MIN for row types”, conforming SQL language shall not contain a <computational operation> that immediately contains MAX or MIN in which the declared type of the <value expression> is a row type.

NOTE 491 — If DISTINCT is specified, then the Conformance Rules of Subclause 9.10, “Grouping operations”, also apply. If MAX or MIN is specified, then the Conformance Rules of Subclause 9.12, “Ordering operations”, also apply.

156) Specifications for Feature T053, “Explicit aliases for all-fields reference”:

a) Subclause 7.12, “<query specification>”:

i) Without Feature T053, “Explicit aliases for all-fields reference”, conforming SQL language shall not contain an <all fields column name list>.

NOTE 492 — If a <set quantifier> DISTINCT is specified, then the Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

157) Specifications for Feature T061, “UCS support”:

a) Subclause 6.1, “<data type>”:

i) Without Feature T061, “UCS support”, conforming SQL language shall not contain a <char length units>.

b) Subclause 6.29, “<string value function>”:

i) Without Feature T061, “UCS support”, conforming SQL language shall not contain a <normalize function>.

c) Subclause 8.12, “<normalized predicate>”:

i) Without Feature T061, “UCS support”, conforming SQL language shall not contain a <normalized predicate>.

158) Specifications for Feature T071, “BIGINT data type”:

a) Subclause 6.1, “<data type>”:

i) Without Feature T071, “BIGINT data type”, conforming SQL language shall not contain BIGINT.

b) Subclause 21.3, “<embedded SQL Ada program>”:

i) Without Feature T071, “BIGINT data type”, conforming SQL language shall not contain an <Ada qualified type specification> that contains Interfaces.SQL.BIGINT.

ii) Without Feature T071, “BIGINT data type”, conforming SQL language shall not contain an <Ada unqualified type specification> that contains BIGINT.

c) Subclause 21.4, “<embedded SQL C program>”:

i) Without Feature T071, “BIGINT data type”, conforming SQL language shall not contain a <C numeric variable> that contains `long long`.

159) Specifications for Feature T111, “Updatable joins, unions and columns”:

a) Subclause 11.22, “<view definition>”:

i) Without Feature T111, “Updatable joins, unions and columns”, in conforming SQL language, if WITH CHECK OPTION is specified, then the viewed table shall be simply updatable.

b) Subclause 14.3, “<cursor specification>”:

i) Without Feature T111, “Updatable joins, unions, and columns”, in conforming SQL language, if FOR UPDATE is specified, then *QE* shall be simply updatable.

c) Subclause 14.9, “<delete statement: searched>”:

i) Without Feature T111, “Updatable joins, unions, and columns”, conforming SQL language shall not contain a <delete statement: searched> that contains a <target table> that identifies a table that is not simply updatable.

d) Subclause 14.10, “<insert statement>”:

i) Without Feature T111, “Updatable joins, unions, and columns”, conforming SQL language shall not contain an <insert statement> that contains an <insertion target> that identifies a table that is not simply updatable.

e) Subclause 14.11, “<merge statement>”:

i) Without Feature T111, “Updatable joins, unions, and columns”, conforming SQL language shall not contain a <merge statement> that contains an <target table> that identifies a table that is not simply updatable.

f) Subclause 14.13, “<update statement: searched>”:

i) Without Feature T111, “Updatable joins, unions, and columns”, conforming SQL language shall not contain an <update statement: searched> that contains a <target table> that identifies a table that is not simply updatable.

160) Specifications for Feature T121, “WITH (excluding RECURSIVE) in query expression”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature T121, “WITH (excluding RECURSIVE) in query expression”, conforming SQL language shall not contain a <query name>.

b) Subclause 7.6, “<table reference>”:

i) Without Feature T121, “WITH (excluding RECURSIVE) in query expression”, conforming SQL language shall not contain a <query name>.

c) Subclause 7.13, “<query expression>”:

i) Without Feature T121, “WITH (excluding RECURSIVE) in query expression”, in conforming SQL language, a <query expression> shall not contain a <with clause>.

161) Specifications for Feature T122, “WITH (excluding RECURSIVE) in subquery”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature T122, “WITH (excluding RECURSIVE) in subquery”, in conforming SQL language, a <query expression> contained in a <query expression> shall not contain a <with clause>.

162) Specifications for Feature T131, “Recursive query”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature T131, “Recursive query”, conforming SQL language shall not contain a <query expression> that contains RECURSIVE.

b) Subclause 11.22, “<view definition>”:

i) Without Feature T131, “Recursive query”, conforming SQL language shall not contain a <view definition> that immediately contains RECURSIVE.

163) Specifications for Feature T132, “Recursive query in subquery”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature T132, “Recursive query in subquery”, in conforming SQL language, a <query expression> contained in a <query expression> shall not contain RECURSIVE.

164) Specifications for Feature T141, “SIMILAR predicate”:

a) Subclause 8.6, “<similar predicate>”:

i) Without Feature T141, “SIMILAR predicate”, conforming SQL language shall not contain a <similar predicate>.

165) Specifications for Feature T151, “DISTINCT predicate”:

a) Subclause 8.15, “<distinct predicate>”:

i) Without Feature T151, “DISTINCT predicate”, conforming SQL language shall not contain a <distinct predicate>.

NOTE 493 — The Conformance Rules of Subclause 9.9, “Equality operations”, also apply.

166) Specifications for Feature T152, “DISTINCT predicate with negation”:

a) Subclause 8.15, “<distinct predicate>”:

i) Without Feature T152, “DISTINCT predicate with negation”, conforming SQL language shall not contain a <distinct predicate part 2> that immediately contains NOT.

167) Specifications for Feature T171, “LIKE clause in table definition”:

a) Subclause 11.3, “<table definition>”:

i) Without Feature T171, “LIKE clause in table definition”, conforming SQL language shall not contain a <like clause>.

168) Specifications for Feature T172, “AS subquery clause in table definition”:

a) Subclause 11.3, “<table definition>”:

i) Without Feature T172, “AS subquery clause in table definition”, conforming SQL language shall not contain an <as subquery clause>.

169) Specifications for Feature T173, “Extended LIKE clause in table definition”:

a) Subclause 11.3, “<table definition>”:

i) Without Feature T173, “Extended LIKE clause in table definition”, a <like clause> shall not contain <like options>.

170) Specifications for Feature T174, “Identity columns”:

a) Subclause 11.4, “<column definition>”:

i) Without Feature T174, “Identity columns”, conforming SQL language shall not contain an <identity column specification>.

b) Subclause 11.17, “<alter identity column specification>”:

i) Without Feature T174, “Identity columns”, an <alter column definition> shall not contain an <alter identity column specification>.

171) Specifications for Feature T175, “Generated columns”:

a) Subclause 11.4, “<column definition>”:

i) Without Feature T175, “Generated columns”, conforming SQL language shall not contain a <generation clause>.

172) Specifications for Feature T176, “Sequence generator support”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature T176, “Sequence generator support”, conforming SQL language shall not contain a <sequence generator name>.

b) Subclause 6.13, “<next value expression>”:

i) Without Feature T176, “Sequence generator support”, conforming SQL language shall not contain a <next value expression>.

c) Subclause 11.62, “<sequence generator definition>”:

i) Without Feature T176, “Sequence generator support”, conforming SQL language shall not contain a <sequence generator definition>.

d) Subclause 11.63, “<alter sequence generator statement>”:

i) Without Feature T176, “Sequence generator support”, conforming SQL language shall not contain an <alter sequence generator statement>.

e) Subclause 11.64, “<drop sequence generator statement>”:

i) Without Feature T176, “Sequence generator support”, conforming SQL language shall not contain a <drop sequence generator statement>.

173) Specifications for Feature T191, “Referential action RESTRICT”:

a) Subclause 11.8, “<referential constraint definition>”:

i) Without Feature T191, “Referential action RESTRICT”, conforming SQL language shall not contain a <referential action> that contains RESTRICT.

174) Specifications for Feature T201, “Comparable data types for referential constraints”:

a) Subclause 11.8, “<referential constraint definition>”:

i) Without Feature T201, “Comparable data types for referential constraints”, conforming SQL language shall not contain a <referencing columns> in which the data type of each referencing column is not the same as the data type of the corresponding referenced column.

NOTE 494 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

175) Specifications for Feature T211, “Basic trigger capability”:

a) Subclause 7.6, “<table reference>”:

i) Without Feature T211, “Basic trigger capability”, conforming SQL language shall not contain a <transition table name>.

b) Subclause 11.39, “<trigger definition>”:

i) Without Feature T211, “Basic trigger capability”, conforming SQL language shall not contain a <trigger definition>.

c) Subclause 11.40, “<drop trigger statement>”:

i) Without Feature T211, “Basic trigger capability”, conforming SQL language shall not contain a <drop trigger statement>.

d) Subclause 12.3, “<privileges>”:

i) Without Feature T211, “Basic trigger capability”, conforming SQL language shall not contain an <action> that contains TRIGGER.

176) Specifications for Feature T212, “Enhanced trigger capability”:

a) Subclause 11.39, “<trigger definition>”:

i) Without Feature T212, “Enhanced trigger capability”, in conforming SQL language, a <triggered action> shall contain FOR EACH ROW.

177) Specifications for Feature T231, “Sensitive cursors”:

a) Subclause 14.2, “<cursor properties>”:

i) Without Feature T231, “Sensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains SENSITIVE.

ii) Without Feature F791, “Insensitive cursors”, or Feature T231, “Sensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains ASENSITIVE.

178) Specifications for Feature T241, “START TRANSACTION statement”:

a) Subclause 17.1, “<start transaction statement>”:

i) Without Feature T241, “START TRANSACTION statement”, conforming SQL language shall not contain a <start transaction statement>.

179) Specifications for Feature T251, “SET TRANSACTION statement: LOCAL option”:

a) Subclause 17.2, “<set transaction statement>”:

i) Without Feature T251, “SET TRANSACTION statement: LOCAL option”, conforming SQL language shall not contain a <set transaction statement> that immediately contains LOCAL.

180) Specifications for Feature T261, “Chained transactions”:

a) Subclause 17.7, “<commit statement>”:

i) Without Feature T261, “Chained transactions”, conforming SQL language shall not contain a <commit statement> that immediately contains CHAIN.

b) Subclause 17.8, “<rollback statement>”:

i) Without Feature T261, “Chained transactions”, conforming SQL language shall not contain a <rollback statement> that immediately contains CHAIN.

181) Specifications for Feature T271, “Savepoints”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature T271, “Savepoints”, conforming SQL language shall not contain a <savepoint name>.

b) Subclause 17.5, “<savepoint statement>”:

i) Without Feature T271, “Savepoints”, conforming SQL language shall not contain a <savepoint statement>.

c) Subclause 17.6, “<release savepoint statement>”:

i) Without Feature T271, “Savepoints”, conforming SQL language shall not contain a <release savepoint statement>.

d) Subclause 17.8, “<rollback statement>”:

i) Without Feature T271, “Savepoints”, conforming SQL language shall not contain a <savepoint clause>.

182) Specifications for Feature T272, “Enhanced savepoint management”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T272, “Enhanced savepoint management”, conforming SQL language shall not contain a <routine characteristics> that contains a <savepoint level indication>.

183) Specifications for Feature T281, “SELECT privilege with column granularity”:

a) Subclause 12.3, “<privileges>”:

i) Without Feature T281, “SELECT privilege with column granularity”, in conforming SQL language, an <action> that contains SELECT shall not contain a <privilege column list>.

184) Specifications for Feature T301, “Functional dependencies”:

a) Subclause 7.10, “<having clause>”:

i) Without Feature T301, “Functional dependencies”, in conforming SQL language, each column reference directly contained in the <search condition> shall be one of the following:

1) An unambiguous reference to a grouping column of *T*.

2) An outer reference.

ii) Without Feature T301, “Functional dependencies”, in conforming SQL language, each column reference contained in a <query expression> in the <search condition> that references a column of *T* shall be one of the following:

1) An unambiguous reference to a grouping column of *T*.

2) Contained in an aggregated argument of a <set function specification>.

b) Subclause 7.11, “<window clause>”:

i) Without Feature T301, “Functional dependencies”, in conforming SQL language, if *T* is a grouped table, then each column reference contained in <window clause> that references a column of *T* shall be a reference to a grouping column of *T* or be contained in an aggregated argument of a <set function specification>.

c) Subclause 7.12, “<query specification>”:

i) Without Feature T301, “Functional dependencies”, in conforming SQL language, if *T* is a grouped table, then in each <value expression> contained in the <select list>, each <column reference> that references a column of *T* shall reference a grouping column or be specified in an aggregated argument of a <set function specification>.

d) Subclause 20.4, “<get descriptor statement>”:

i) Without Feature T301, “Functional dependencies”, conforming SQL language shall not contain a <descriptor item name> that contains KEY_MEMBER.

185) Specifications for Feature T312, “OVERLAY function”:

a) Subclause 6.29, “<string value function>”:

i) Without Feature T312, “OVERLAY function”, conforming SQL language shall not contain a <character overlay function>.

ii) Without Feature T312, “OVERLAY function”, conforming SQL language shall not contain a <binary overlay function>.

186) Specifications for Feature T322, “Overloading of SQL-invoked functions and procedures”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T322, “Overloading of SQL-invoked functions and procedures”, conforming SQL language shall not contain a <schema routine> in which the schema identified by the explicit or implicit schema name of the <schema qualified routine name> includes a routine descriptor whose routine name is <schema qualified routine name>.

187) Specifications for Feature T323, “Explicit security for external routines”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T323, “Explicit security for external routines”, conforming SQL language shall not contain an <external security clause>.

188) Specifications for Feature T324, “Explicit security for SQL routines”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T324, “Explicit security for SQL routines”, conforming SQL language shall not contain a <rights clause>.

189) Specifications for Feature T325, “Qualified SQL parameter references”:

a) Subclause 6.6, “<identifier chain>”:

i) Without Feature T325, “Qualified SQL parameter references”, conforming SQL language shall not contain an SQL parameter reference whose first <identifier> is the <qualified identifier> of a <routine name>.

b) Subclause 7.12, “<query specification>”:

i) Without Feature T325, “Qualified SQL parameter references”, conforming SQL language shall not contain an <asterisked identifier chain> whose referent is an SQL parameter and whose first <identifier> is the <qualified identifier> of a <routine name>.

190) Specifications for Feature T326, “Table functions”:

a) Subclause 6.39, “<multiset value constructor>”:

i) Without Feature T326, “Table functions”, a <multiset value constructor> shall not contain a <table value constructor by query>.

b) Subclause 7.6, “<table reference>”:

i) Without Feature T326, “Table functions”, conforming SQL language shall not contain a <table function derived table>.

c) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T326, “Table functions”, conforming SQL language shall not contain a <returns table type>.

191) Specifications for Feature T331, “Basic roles”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature T331, “Basic roles”, conforming SQL language shall not contain a <role name>.

b) Subclause 12.4, “<role definition>”:

i) Without Feature T331, “Basic roles”, conforming SQL language shall not contain a <role definition>.

c) Subclause 12.5, “<grant role statement>”:

i) Without Feature T331, “Basic roles”, conforming SQL language shall not contain a <grant role statement>.

d) Subclause 12.6, “<drop role statement>”:

i) Without Feature T331, “Basic roles”, conforming SQL language shall not contain a <drop role statement>.

e) Subclause 12.7, “<revoke statement>”:

i) Without Feature T331, “Basic roles”, conforming SQL language shall not contain a <revoke role statement>.

f) Subclause 19.3, “<set role statement>”:

i) Without Feature T331, “Basic roles”, conforming SQL language shall not contain a <set role statement>.

192) Specifications for Feature T332, “Extended roles”:

a) Subclause 6.4, “<value specification> and <target specification>”:

i) Without Feature T332, “Extended roles”, conforming SQL language shall not contain CURRENT_ROLE.

b) Subclause 11.5, “<default clause>”:

i) Without Feature T332, “Extended roles”, conforming SQL language shall not contain a <default option> that contains CURRENT_ROLE.

c) Subclause 12.3, “<privileges>”:

i) Without Feature T332, “Extended roles”, conforming SQL language shall not contain a <grantor>.

d) Subclause 12.4, “<role definition>”:

i) Without Feature T332, “Extended roles”, conforming SQL language shall not contain a <role definition> that immediately contains WITH ADMIN.

e) Subclause 12.8, “Grantor determination”:

i) Without Feature T332, “Extended roles”, conforming SQL language shall contain no <grantor>.

193) Specifications for Feature T351, “Bracketed comments”:

a) Subclause 5.2, “<token> and <separator>”:

i) Without Feature T351, “Bracketed comments”, conforming SQL language shall not contain a <bracketed comment>.

194) Specifications for Feature T431, “Extended grouping capabilities”:

a) Subclause 6.9, “<set function specification>”:

i) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain a <grouping operation>.

b) Subclause 7.9, “<group by clause>”:

i) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain a <rollup list>.

ii) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain a <cube list>.

iii) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain a <grouping sets specification>.

iv) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain an <empty grouping set>.

v) Without Feature T431, “Extended grouping capabilities”, conforming SQL language shall not contain an <ordinary grouping set> that contains a <grouping column reference list>.

195) Specifications for Feature T432, “Nested and concatenated GROUPING SETS”:

a) Subclause 7.9, “<group by clause>”:

i) Without Feature T432, “Nested and concatenated GROUPING SETS”, conforming SQL language shall not contain a <grouping set list> that contains a <grouping sets specification>.

ii) Without Feature T432, “Nested and concatenated GROUPING SETS”, conforming SQL language shall not contain a <group by clause> that simply contains a <grouping sets specification> *GSS* where *GSS* is not the only <grouping element> simply contained in the <group by clause>.

NOTE 495 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

196) Specifications for Feature T433, “Multiargument GROUPING function”:

a) Subclause 6.9, “<set function specification>”:

i) Without Feature T433, “Multiargument GROUPING function”, conforming SQL language shall not contain a <grouping operation> that contains more than one <column reference>.

197) Specifications for Feature T434, “GROUP BY DISTINCT”:

a) Subclause 7.9, “<group by clause>”:

i) Without Feature T434, “GROUP BY DISTINCT”, conforming SQL language shall not contain a <group by clause> that simply contains a <set quantifier>.

198) Specifications for Feature T441, “ABS and MOD functions”:

a) Subclause 6.27, “<numeric value function>”:

i) Without Feature T441, “ABS and MOD functions”, conforming language shall not contain an <absolute value expression>.

ii) Without Feature T441, “ABS and MOD functions”, conforming language shall not contain a <modulus expression>.

199) Specifications for Feature T461, “Symmetric BETWEEN predicate”:

a) Subclause 8.3, “<between predicate>”:

i) Without Feature T461, “Symmetric BETWEEN predicate”, conforming SQL language shall not contain SYMMETRIC or ASYMMETRIC.

NOTE 496 — Since <between predicate> is an ordering operation, the Conformance Rules of Subclause 9.12, “Ordering operations”, also apply.

200) Specifications for Feature T471, “Result sets return value”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T471, “Result sets return value”, conforming SQL language shall not contain a <returned result sets characteristic>.

b) Subclause 14.2, “<cursor properties>”:

i) Without Feature T471, “Result sets return value”, conforming SQL language shall not contain a <cursor returnability>.

c) Subclause 20.15, “<allocate cursor statement>”:

i) Without Feature T471, “Result sets return value”, conforming SQL language shall not contain a <result set cursor>.

201) Specifications for Feature T491, “LATERAL derived table”:

a) Subclause 7.6, “<table reference>”:

i) Without Feature T491, “LATERAL derived table”, conforming SQL language shall not contain a <lateral derived table>.

202) Specifications for Feature T501, “Enhanced EXISTS predicate”:

a) Subclause 8.10, “<exists predicate>”:

i) Without Feature T501, “Enhanced EXISTS predicate”, conforming SQL language shall not contain an <exists predicate> that simply contains a <table subquery> in which the <select list> of a <query specification> directly contained in the <table subquery> does not comprise either an <asterisk> or a single <derived column>.

203) Specifications for Feature T511, “Transaction counts”:

a) Subclause 23.1, “<get diagnostics statement>”:

i) 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.

204) Specifications for Feature T551, “Optional key words for default syntax”:

a) Subclause 7.13, “<query expression>”:

i) Without Feature T551, “Optional key words for default syntax”, conforming SQL language shall not contain UNION DISTINCT, EXCEPT DISTINCT, or INTERSECT DISTINCT.

b) Subclause 14.2, “<cursor properties>”:

i) Without Feature T551, “Optional key words for default syntax”, conforming SQL language shall not contain a <cursor holdability> that immediately contains WITHOUT HOLD.

205) Specifications for Feature T561, “Holdable locators”:

a) Subclause 14.16, “<free locator statement>”:

i) Without Feature T561, “Holdable locators”, conforming SQL language shall not contain a <free locator statement>.

b) Subclause 14.17, “<hold locator statement>”:

i) Without Feature T561, “Holdable locators”, conforming SQL language shall not contain a <hold locator statement>.

206) Specifications for Feature T571, “Array-returning external SQL-invoked functions”:

a) Subclause 11.41, “<user-defined type definition>”:

i) Without Feature T571, “Array-returning external SQL-invoked functions”, conforming SQL language shall not contain a <method specification> that contains a <returns clause> that satisfies either of the following conditions:

1) A <result cast from type> is specified that simply contains an <array type> and does not contain a <locator indication>.

2) A <result cast from type> is not specified and <returns data type> simply contains an <array type> and does not contain a <locator indication>.

b) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T571, “Array-returning external SQL-invoked functions”, conforming SQL language shall not contain an <SQL-invoked routine> that defines an array-returning external function.

207) Specifications for Feature T572, “Multiset-returning external SQL-invoked functions”:

a) Subclause 11.41, “<user-defined type definition>”:

i) Without Feature T572, “Multiset-returning external SQL-invoked functions”, conforming SQL language shall not contain a <method specification> that contains a <returns clause> that satisfies either of the following conditions:

1) A <result cast from type> is specified that simply contains a <multiset type> and does not contain a <locator indication>.

2) A <result cast from type> is not specified and <returns data type> simply contains a <multiset type> and does not contain a <locator indication>.

b) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T572, “Multiset-returning external SQL-invoked functions”, conforming SQL language shall not contain an <SQL-invoked routine> that defines a multiset-returning external function.

208) Specifications for Feature T581, “Regular expression substring function”:

a) Subclause 6.29, “<string value function>”:

i) Without Feature T581, “Regular expression substring function”, conforming SQL language shall not contain a <regular expression substring function>.

209) Specifications for Feature T591, “UNIQUE constraints of possibly null columns”:

a) Subclause 11.7, “<unique constraint definition>”:

i) Without Feature T591, “UNIQUE constraints of possibly null columns”, in conforming SQL language, if UNIQUE is specified, then the <column definition> for each column whose <column name> is contained in the <unique column list> shall contain NOT NULL.

NOTE 497 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

210) Specifications for Feature T601, “Local cursor references”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature T601, “Local cursor references”, a <cursor name> shall not contain a <local qualifier>.

211) Specifications for Feature T611, “Elementary OLAP operations”:

a) Subclause 6.10, “<window function>”:

i) Without Feature T611, “Elementary OLAP operations”, conforming SQL language shall not contain a <window function>.

b) Subclause 7.11, “<window clause>”:

i) Without Feature T611, “Elementary OLAP operations”, conforming SQL language shall not contain a <window specification>.

c) Subclause 10.10, “<sort specification list>”:

i) Without Feature T611, “Elementary OLAP operations”, conforming SQL language shall not contain a <null ordering>.

NOTE 498 — The Conformance Rules of Subclause 9.12, “Ordering operations”, also apply.

212) Specifications for Feature T612, “Advanced OLAP operations”:

a) Subclause 5.4, “Names and identifiers”:

i) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window name>.

b) Subclause 6.10, “<window function>”:

i) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window name>.

ii) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain PERCENT_RANK or CUME_DIST.

iii) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window function> that simply contains ROW_NUMBER and immediately contains a <window name or specification> whose window structure descriptor does not contain a window ordering clause.

c) Subclause 6.27, “<numeric value function>”:

i) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <width bucket function>.

d) Subclause 7.11, “<window clause>”:

i) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window clause>.

ii) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain an <existing window name>.

iii) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <window frame exclusion>.

NOTE 499 — The Conformance Rules of Subclause 9.10, “Grouping operations”, also apply.

e) Subclause 10.9, “<aggregate function>”:

i) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <hypothetical set function> or an <inverse distribution function>.

ii) Without Feature T612, “Advanced OLAP operations”, conforming SQL language shall not contain a <filter clause>.

213) Specifications for Feature T613, “Sampling”:

a) Subclause 7.6, “<table reference>”:

i) Without Feature T613, “Sampling”, conforming SQL language shall not contain a <sample clause>.

214) Specifications for Feature T621, “Enhanced numeric functions”:

a) Subclause 6.27, “<numeric value function>”:

i) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <natural logarithm>.

ii) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain an <exponential function>.

iii) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <power function>.

iv) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <square root>.

v) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <floor function>.

vi) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <ceiling function>.

b) Subclause 10.9, “<aggregate function>”:

i) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <computational operation> that immediately contains STDDEV_POP, STDDEV_SAMP, VAR_POP, or VAR_SAMP.

ii) Without Feature T621, “Enhanced numeric functions”, conforming SQL language shall not contain a <binary set function type>.

215) Specifications for Feature T641, “Multiple column assignment”:

a) Subclause 14.14, “<set clause list>”:

i) Without Feature T641, “Multiple column assignment”, conforming SQL language shall not contain a <multiple column assignment>.

216) Specifications for Feature T651, “SQL-schema statements in SQL routines”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T651, “SQL-schema statements in SQL routines”, conforming SQL language shall not contain an <SQL routine body> that contains an SQL-schema statement.

217) Specifications for Feature T652, “SQL-dynamic statements in SQL routines”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T652, “SQL-dynamic statements in SQL routines”, conforming SQL language shall not contain an <SQL routine body> that contains an SQL-dynamic statement.

218) Specifications for Feature T653, “SQL-schema statements in external routines”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T653, “SQL-schema statements in external routines”, conforming SQL language shall not contain an <external routine name> that identifies a program in which an SQL-schema statement appears.

219) Specifications for Feature T654, “SQL-dynamic statements in external routines”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T654, “SQL-dynamic statements in external routines”, conforming SQL language shall not contain an <external routine name> that identifies a program in which an SQL-dynamic statement appears.

220) Specifications for Feature T655, “Cyclically dependent routines”:

a) Subclause 11.50, “<SQL-invoked routine>”:

i) Without Feature T655, “Cyclically dependent routines”, conforming SQL language shall not contain an <SQL routine body> that contains a <routine invocation> whose subject routine is generally dependent on the routine descriptor of the SQL-invoked routine specified by <SQL-invoked routine>.

Annex B
(informative)

Implementation-defined elements

This Annex is modified by Annex C, "Implementation-defined elements", in ISO/IEC 9075-3.
This Annex is modified by Annex B, "Implementation-defined elements", in ISO/IEC 9075-4.
This Annex is modified by Annex B, "Implementation-defined elements", in ISO/IEC 9075-9.
This Annex is modified by Annex B, "Implementation-defined elements", in ISO/IEC 9075-10.
This Annex is modified by Annex B, "Implementation-defined elements", in ISO/IEC 9075-11.
This Annex is modified by Annex B, "Implementation-defined elements", in ISO/IEC 9075-13.
This Annex is modified by Annex B, "Implementation-defined elements", in ISO/IEC 9075-14.

This Annex references those features that are identified in the body of this part of ISO/IEC 9075 as implementation-defined.

1) Subclause 4.2.2, "Comparison of character strings":

   a) The specific character set associated with the subtype of character string represented by the <key word>s NATIONAL CHARACTER is implementation-defined.

   b) The circumstances in which conversion of non-UCS character string expressions from one character set to another is automatic is implementation-defined.

2) Subclause 4.2.4, "Character repertoires":

   a) The character repertoires, including standard defined repertoires, supported by the SQL-implementation are implementation-defined.

   b) The character set named SQL_TEXT is an implementation-defined character set that contains every character that is in <SQL language character> and all characters that are in other character sets supported by the SQL-implementation.

   c) The character set named SQL_IDENTIFIER is an implementation-defined character set that contains every character that is in <SQL language character> and all characters that the SQL-implementation supports for use in <regular identifier>s, which is the same as the repertoire that the SQL-implementation supports for use in <delimited identifier>s.

3) Subclause 4.2.5, "Character encoding forms":

   a) The character encodings in the SQL_CHARACTER character encoding form are implementation-defined.

   b) The character encodings in the SQL_TEXT character encoding form are implementation-defined.

   c) The character encodings in the SQL_IDENTIFIER character encoding form are implementation-defined.

   d) Which character encoding forms, including standard defined encoding forms, is implemented are implementation-defined.

4) Subclause 4.2.6, “Collations”:

   a) The collations, including standard defined collations, supported by the SQL-implementation are implementation-defined.

   b) The ordering specified by the SQL_CHARACTER collation is implementation-defined.

   c) The ordering specified by the SQL_TEXT collation is implementation-defined.

   d) The ordering specified by the SQL_IDENTIFIER collation is implementation-defined.

5) Subclause 4.2.7, “Character sets”:

   a) It is implementation-defined which collation, UCS_BASIC or UNICODE, is the default for the UTF8, UTF16, and UTF32 character sets.

6) Subclause 4.2.8, “Universal character sets”:

   a) With the exception of <normalize function> and <normalized predicate>, the result of any operation on an unnormalized UCS string is implementation-defined.

7) Subclause 4.3.2, “Binary string comparison”:

   a) For binary string values other than binary large object string values, it is implementation-defined whether trailing X’00’s are considered significant when comparing two binary string values that are otherwise equivalent.

8) Subclause 4.4, “Numbers”:

   a) Whether truncation or rounding is performed when trailing digits are removed from a numeric value is implementation-defined.

   b) When an approximation is obtained by truncation or rounding and there are more than one approximation, then it is implementation-defined which approximation is chosen.

   c) It is implementation-defined which numeric values have approximations obtained by rounding or truncation for a given approximate numeric type.

   d) The boundaries within which the normal rules of arithmetic apply are implementation-defined.

9) Subclause 4.4.2, “Characteristics of numbers”:

   a) When converting between numeric data types, if least significant digits are lost, then it is implementation-defined whether rounding or truncation occurs.

10) Subclause 4.6.2, “Datetimes”:

   a) Whether an SQL-implementation supports leap seconds, and the consequences of such support for date and interval arithmetic, are implementation-defined.

11) Subclause 4.9, “Reference types”:

   a) In a host variable, a reference type is materialized as an *n*-octet value, where *N* is implementation-defined.

12) Subclause 4.14.6, “Operations involving tables”:

a) If a <table reference> contains a <sample clause>, and the <sample clause> contains <repeatable clause>, then repeated executions of that <table reference> return a result table with identical rows for a given <repeat argument>, provided certain implementation-defined conditions are satisfied.

13) Subclause 4.15.3, “Window functions”:

a) If PERCENT_RANK is specified, then the declared type of the result is approximate numeric with implementation-defined precision.

b) If CUME_DIST is specified, then the declared type of the result is approximate numeric with implementation-defined precision.

14) Subclause 4.17.3, “Table constraints”:

a) The ordering of the lists of referencing column names and referenced column names in a referential constrict descriptor is implementation-defined, but shall be such that corresponding column names occupy corresponding positions in each list.

15) Subclause 4.18, “Functional dependencies”:

a) An SQL-implementation may define additional known functional dependencies.

16) Subclause 4.22, “SQL-client modules”:

a) The mechanisms by which SQL-client modules are created or destroyed are implementation-defined.

b) The manner in which an association between an SQL-client module and an SQL-agent is defined is implementation-defined.

c) Whether a compilation unit may invoke or transfer control to other compilation units, written in the same or a different programming language is implementation-defined.

17) Subclause 4.23, “Embedded syntax”:

a) Whether a portion of the name space is reserved by an implementation for the names of procedures, subroutines, program variables, branch labels, <SQL-client module definition>s, or <externally-invoked procedure>s is implementation-defined; if a portion of the name space is so reserved, the portion reserved is also implementation-defined.

18) Subclause 4.24, “Dynamic SQL concepts”:

a) Within an SQL-session, all prepared statements belong to the same implementation-defined <SQL-client module definition> that is different from any other <SQL-client module definition> that exists simultaneously in the environment.

19) Subclause 4.25, “Direct invocation of SQL”:

a) The method of invoking <direct SQL statement>s, the method of raising conditions as a result of <direct SQL statement>s, the method of accessing diagnostic information, and the method of returning the results are all implementation-defined.

20) Subclause 4.34, “Basic security model”:

a) The mapping of <authorization identifier>s to operating system users is implementation-defined.

21) Subclause 4.32, “Cursors”:

a) If a sensitive or asensitive holdable cursor is held open for a subsequent SQL-transaction, then whether any significant changes made to SQL-data (by this or any subsequent SQL-transaction in which the cursor is held open) will be visible through that cursor in the subsequent SQL-transaction before that cursor is closed is implementation-defined.

22) Subclause 4.35, "SQL-transactions":

a) It is implementation-defined whether or not the execution of an SQL-data statement is permitted to occur within the same SQL-transaction as the execution of an SQL-schema statement. If it does occur, then the effect on any open cursor or deferred constraint is also implementation-defined.

b) If an SQL-implementation detects unrecoverable errors and implicitly initiates the execution of a <rollback statement>, an exception condition is raised with an *implementation-defined exception code*.

c) It is implementation-defined whether or not the dynamic execution of an <SQL dynamic data statement> is permitted to occur within the same SQL-transaction as the dynamic execution of an SQL-schema statement. If it does occur, then the effect on any prepared dynamic statement is also implementation-defined.

d) It is implementation-defined whether or not, or how, a <rollback statement> that references a <savepoint specifier> affects diagnostics area contents, the contents of SQL descriptor areas, and the status of prepared statements.

23) Subclause 4.36, "SQL-connections":

a) It is implementation-defined how an SQL-implementation uses <SQL-server name> to determine the location, identity, and communication protocol required to access the SQL-server and initiate an SQL-session.

24) Subclause 4.37, "SQL-sessions":

a) When an SQL-session is initiated other than through the use of an explicit <connect statement>, then an SQL-session associated with an implementation-defined SQL-server is initiated. The default SQL-server is implementation-defined.

b) The mechanism and rules by which an SQL-implementation determines whether a call to an <externally-invoked procedure> is the last call within the last active SQL-client module is implementation-defined.

c) An SQL-session uses one or more implementation-defined schemas that contain the instances of any global temporary tables, created local temporary tables, or declared local temporary tables within the SQL-session.

d) When an SQL-session is initiated, there is an implementation-defined default time zone used as the current default time zone displacement of the SQL-session.

e) When an SQL-session is initiated other than through the use of an explicit <connect statement>, there is an implementation-defined initial value of the SQL-session user identifier.

f) When an SQL-session is initiated, there is an implementation-defined default catalog whose name is used to effectively qualify all unqualified <schema name>s contained in <preparable statement>s that are dynamically prepared in the current SQL-session through the execution of <prepare statement>s and <execute immediate statement>s.

g) When an SQL-session is initiated, there is an implementation-defined default schema whose name is used to effectively qualify all unqualified <schema qualified name>s contained in <preparable state-

ment>s that are dynamically prepared in the current SQL-session through the execution of <prepare statement>s and <execute immediate statement>s.

h) The value of the current SQL-path before a successful execution of <set path statement> is implementation-defined.

25) Subclause 5.1, "<SQL terminal character>":

a) The end-of-line indicator (<newline>) is implementation-defined.

26) Subclause 5.2, "<token> and <separator>":

a) Equivalence of two <regular identifier>s, or of a <regular identifier> and a <delimited identifier> is determined using an implementation-defined collation that is sensitive to case.

b) When the source character set does not contain <reverse solidus>, the character used as the default <Unicode escape character> is implementation-defined.

27) Subclause 5.3, "<literal>":

a) The <character set name> character set of the used to represent national characters is implementation-defined.

b) The declared type of an <exact numeric literal> is implementation-defined.

c) The declared type of an <approximate numeric literal> is implementation-defined.

d) The declared type of a <binary string literal> is implementation-defined.

28) Subclause 5.4, "Names and identifiers":

a) If a <schema name> contained in a <schema name clause> but not contained in an SQL-client module does not contain a <catalog name>, then an implementation-defined <catalog name> is implicit.

b) If a <schema name> contained in a <module authorization clause> does not contain a <catalog name>, then an implementation-defined <catalog name> is implicit.

c) Those <identifier>s that are valid <authorization identifier>s are implementation-defined.

d) Those <identifier>s that are valid <catalog name>s are implementation-defined.

e) All <transcoding name>s are implementation-defined.

f) If a <schema name> contained in a <preparable statement> that is dynamically prepared in the current SQL-session through the execution of a <prepare statement> or an <execute immediate statement> does not contain a <catalog name>, then the implementation-defined <catalog name> for the SQL-session is implicit.

g) If a <schema qualified name> contained in a <preparable statement> that is dynamically prepared in the current SQL-session through the execution of a <prepare statement> or an <execute immediate statement> does not contain a <schema name>, then the implementation-defined <schema name> for the SQL-session is implicit.

29) Subclause 6.1, "<data type>":

a) The <character set name> associated with NATIONAL CHARACTER is implementation-defined.

b) If a <precision> is omitted, then an implementation-defined <precision> is implicit.

c) The decimal precision of a data type defined as DECIMAL for each value specified by <precision> is implementation-defined.

d) The precisions of data types defined as SMALLINT, INTEGER, and BIGINT are implementation-defined, but all three data types have the same radix.

e) The binary precision of a data type defined as FLOAT for each value specified by <precision> is implementation-defined.

f) The precision of a data type defined as REAL is implementation-defined.

g) The precision of a data type defined as DOUBLE PRECISION is implementation-defined, but greater than that for REAL.

h) For every <data type>, the limits of the <data type> are implementation-defined.

i) The maximum lengths for character string types and binary string types are implementation-defined.

j) If CHARACTER SET is not specified for <character string type>, then the character set is implementation-defined.

k) For the <exact numeric type>s DECIMAL and NUMERIC, the maximum values of <precision> and of <scale> are implementation-defined.

l) The transformation *ENNF*() of an <exact numeric type> to its normal form, to obtain the data type name saved in a numeric data type descriptor, is implementation-defined, though it shall adhere to the following constraints:

   i) For every <exact numeric type> *ENT*, *ENNF*(*ENT*) shall not specify DEC or INT.

      NOTE 500 — The preceding requirement prohibits the function *ENNF* from returning a value that uses the abbreviated spelling of the two data types; the function shall instead return the long versions of DECIMAL or INTEGER.

   ii) For every <exact numeric type> *ENT*, the precision, scale, and radix of *ENNF*(*ENT*) shall be the precision, scale, and radix of *ENT*.

   iii) For every <exact numeric type> *ENT*, *ENNF*(*ENT*) shall be the same as *ENNF*(*ENNF*(*ENT*)).

   iv) For every <exact numeric type> *ENT*, if *ENNF*(*ENT*) specifies DECIMAL, then *ENNF*(*ENT*) shall specify <precision>, and the precision of *ENNF*(*ENT*) shall be the value of the <precision> specified in *ENNF*(*ENT*).

m) For the <approximate numeric type> FLOAT, the maximum value of <precision> is implementation-defined.

n) The transformation *ANNF*() of an <approximate numeric type> to its normal form, to obtain the data type name saved in a numeric data type descriptor, is implementation-defined, though it shall adhere to the following constraints:

   i) For every <approximate numeric type> *ANT*, the precision of *ANNF*(*ANT*) shall be the precision of *ANT*.

   ii) For every <approximate numeric type> *ANT*, *ANNF*(*ANT*) shall be the same as *ANNF*(*ANNF*(*ANT*)).

   iii) For every <approximate numeric type> *ANT*, if *ANNF*(*ANT*) specifies FLOAT, then *ANNF*(*ANT*) shall specify <precision>, and the precision of *ANNF*(*ANT*) shall be the value of the <precision> specified in *ANNF*(*ANT*).

o) For the <approximate numeric type>s FLOAT, REAL, and DOUBLE PRECISION, the maximum and minimum values of the exponent are implementation-defined.

p) The maximum value of <time fractional seconds precision> is implementation-defined, but shall not be less than 6.

q) The maximum values of <time precision> and <timestamp precision> for a <datetime type> are the same implementation-defined value.

r) If the maximum cardinality of an <array type> is omitted, then an implementation-defined maximum cardinality is implicit.

30) Subclause 6.4, "<value specification> and <target specification>":

a) Whether the character string of the <value specification>s CURRENT_USER, SESSION_USER, and SYSTEM_USER is variable-length or fixed-length, and its maximum length if it is variable-length or its length if it is fixed-length, are implementation-defined.

b) Whether the character string of the <value specification>s CURRENT_CATALOG and CURRENT_SCHEMA is variable-length or fixed-length, and its maximum length if it is variable-length or its length if it is fixed-length, are implementation-defined.

c) The value specified by SYSTEM_USER is an implementation-defined string that represents the operating system user who executed the SQL-client module that contains the SQL-statement whose execution caused the SYSTEM_USER <general value specification> to be evaluated.

d) Whether the data type of CURRENT_PATH is fixed-length or variable-length, and its length if it is fixed-length or its maximum length if it is variable-length, are implementation-defined.

e) If a <target specification> or <simple target specification> is assigned a value that is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string*.

f) Without Feature F611, "Indicator data types", in conforming SQL language, the declared types of <indicator parameter>s and <indicator variable>s shall be the same implementation-defined data type.

31) Subclause 6.9, "<set function specification>":

a) The precision of the value derived from application of the COUNT function is implementation-defined.

b) The precision of the value derived from application of the SUM function to a declared type of exact numeric is implementation-defined.

c) The precision and scale of the value derived from application of the AVG function to a declared type of exact numeric is implementation-defined.

d) The precision of the value derived from application of the SUM function or AVG function to a data type of approximate numeric is implementation-defined.

32) Subclause 6.12, "<cast specification>":

a) Whether to round or truncate when casting to exact numeric, approximate numeric, datetime, or interval data types is implementation-defined.

33) Subclause 6.26, "<numeric value expression>":

a) When the declared type of both operands of the addition, subtraction, multiplication, or division operator is exact numeric, the declared type of the result is an implementation-defined exact numeric type.

b) When the declared type of both operands of the division operator is exact numeric, the scale of the result is implementation-defined.

c) When the declared type of either operand of an arithmetic operator is approximate numeric, the declared type of the result is an implementation-defined approximate numeric type.

d) Whether to round or truncate when performing division is implementation-defined.

34) Subclause 6.27, "<numeric value function>":

a) The declared type of <position expression> is an implementation-defined exact numeric type with scale 0 (zero).

b) The declared type of <extract expression> is an implementation-defined exact numeric type. If <primary datetime field> specifies SECOND, then the scale is implementation-defined; otherwise, the scale is 0 (zero).

c) The declared type of <length expression> is an implementation-defined exact numeric type with scale 0 (zero).

d) The declared type of the result of <natural logarithm> is an implementation-defined approximate numeric.

e) The declared type of the result of <exponential function> is an implementation-defined approximate numeric.

f) The declared type of the result of <power function> is an implementation-defined approximate numeric.

g) The declared types of the results of <floor function> and of <ceiling function> are implementation-defined exact numeric type with scale 0 (zero) if the declared type of the simply contained <numeric value expression> is exact numeric; otherwise, the declared types of the results are implementation-defined approximate numeric types.

h) The declared type of <regex occurrences function> is an implementation-defined exact numeric type with scale 0 (zero).

i) The declared type of <regex position expression> is an implementation-defined exact numeric type with scale 0 (zero).

35) Subclause 6.28, "<string value expression>":

a) If the result of the <character value expression> is a zero-length character string, then it is implementation-defined whether an exception condition is raised: *data exception — zero-length character string*.

b) If the character encoding form of <character factor> is UTF8, UTF16, or UTF32, and either of the operands is not normalized, then the result is implementation-defined.

36) Subclause 6.29, "<string value function>":

a) The maximum length of <character transliteration> or <transcoding> is implementation-defined.

b) The character set of the result of a <transcoding> is implementation-defined.

c) If <normalize function result length> is not specified, then the maximum length of the declared type of the <normalize function> is implementation-defined.

d) If no <XQuery replacement string> is specified, then the maximum length of the declared type of <regex transliteration> is implementation-defined.

37) Subclause 6.32, “<interval value expression>”:

a) The difference of two values of type TIME (with or without time zone) is constrained to be between –24:00:00 and +24:00:00 (excluding each end point); it is implementation-defined which of two non-zero values in this range is the result, although the computation shall be deterministic.

b) When an interval is produced from the difference of two datetimes, the choice of whether to round or truncate is implementation-defined.

c) The result's <interval leading field precision> is implementation-defined, but shall not be less than the <interval leading field precision> of the <interval primary>.

d) The <interval leading field precision> is implementation-defined, but shall be sufficient to represent all interval values with the interval fields and <interval leading field precision> of <interval value expression 1> as well as all interval values with the interval fields and <interval leading field precision> of <interval term 1>.

38) Subclause 7.12, “<query specification>”:

a) An SQL-implementation may define additional implementation-defined rules for recognizing known-not-null columns.

39) Subclause 8.2, “<comparison predicate>”:

a) For two binary string values *X* and *Y*, neither of which is a binary large object string value, it is implementation-defined whether *X* is less than *Y* or whether *X* is equal to *Y*, if the only difference between *X* and *Y* are trailing X’00’s in *Y*.

40) Subclause 8.5, “<like predicate>”:

a) It is implementation-defined which collations can be used as collations for the <like predicate>.

41) Subclause 8.6, “<similar predicate>”:

a) It is implementation-defined which collations can be used as collations for the <similar predicate>.

42) Subclause 9.1, “Retrieval assignment”:

a) If a value *V* is approximate numeric and a target *T* is exact numeric, then whether the approximation of *V* retrieved into *T* is obtained by rounding or by truncation is implementation-defined.

b) If a value *V* is datetime with a greater precision than a target *T*, then it is implementation-defined whether the approximation of *V* retrieved into *T* is obtained by rounding or truncation.

c) If a value *V* is interval with a greater precision than a target *T*, then it is implementation-defined whether the approximation of *V* retrieved into *T* is obtained by rounding or by truncation.

43) Subclause 9.2, “Store assignment”:

a) If a value *V* is approximate numeric and a target *T* is exact numeric, then whether the approximation of *V* stored into *T* is obtained by rounding or by truncation is implementation-defined.

b) If a value *V* is datetime with a greater precision than a target *T*, then it is implementation-defined whether the approximation of *V* stored into *T* is obtained by rounding or truncation.

c) If a value *V* is interval with a greater precision than a target *T*, then it is implementation-defined whether the approximation of *V* stored into *T* is obtained by rounding or by truncation.

44) Subclause 9.3, “Result of data type combinations”:

a) If all of the data types in *DTS* are exact numeric, then the result data type is exact numeric with implementation-defined precision.

b) If any data type in *DTS* is approximate numeric, then each data type in *DTS* shall be numeric and the result data type is approximate numeric with implementation-defined precision.

45) Subclause 9.16, “XQuery regular expression matching”:

a) If the character repertoire of the pattern or flag is not UCS, then the conversion to UCS is implementation-defined.

b) If the character repertoire of the subject string is not UCS, then the definition of “XQuery regular expression match” is implementation-defined.

46) Subclause 9.17, “XQuery regular expression replacement”:

a) If the character repertoire of the pattern, replacement string, or flag is not UCS, then the conversion to UCS is implementation-defined.

b) If the character repertoire of the subject string is not UCS, then the result of the Subclause is implementation-defined.

47) Subclause 9.24, “Creation of a sequence generator”:

a) If <sequence generator maxvalue option> specifies NO MAXVALUE or if <sequence generator maxvalue option> is not specified, then a <sequence generator max value> that is an implementation-defined <signed numeric literal> of declared type *DT* is implicit.

b) If <sequence generator minvalue option> specifies NO MINVALUE or if <sequence generator minvalue option> is not specified, then a <sequence generator min value> that is an implementation-defined <signed numeric literal> of declared type *DT* is implicit.

48) Subclause 9.25, “Altering a sequence generator”:

a) If <sequence generator maxvalue option> specifies NO MAXVALUE, then a <sequence generator max value> that is an implementation-defined <signed numeric literal> of declared type *DT* is implicit.

b) If <sequence generator minvalue option> specifies NO MINVALUE, then a <sequence generator min value> that is an implementation-defined <signed numeric literal> of declared type *DT* is implicit.

49) Subclause 10.11, “Determination of view and view component privileges”:

a) If an <authorization identifier> *A* has UPDATE privilege on every column of a view *V*, then it is implementation-defined whether *A* has UPDATE privilege on *V*.

b) If an <authorization identifier> *A* has INSERT privilege on every column of a view *V*, then it is implementation-defined whether *A* has INSERT privilege on *V*.

50) Subclause 10.1, “<interval qualifier>”:

a) The maximum value of <interval leading field precision> is implementation-defined, but shall not be less than 2.

b) The maximum value of <interval fractional seconds precision> is implementation-defined, but shall not be less than 2.

51) Subclause 10.4, "<routine invocation>":

a) If an SQL-invoked routine does not contain SQL, does not possibly read SQL-data, and does not possibly modify SQL-data, then the SQL-session module of the new SQL-session context *RSC* is set to be an implementation-defined module.

b) If p_i is an output SQL parameter, then CPV_i is an implementation-defined value of type T_i.

c) Whether a syntax error occurs if an <SQL routine body> contains an <SQL connection statement> or an <SQL transaction statement> is implementation-defined.

d) When a new SQL-session context *RSC* is created, the current default catalog name, current default unqualified schema name, the current default character set name, the SQL-path of the current SQL-session, the current default time zone displacement of the current SQL-session, and the contents of all SQL dynamic descriptor areas are set to implementation-defined values.

e) If *R* is an external routine, then it is implementation-defined whether the identities of all instances of created local temporary tables that are referenced in the <SQL-client module definition> of *p*, declared local temporary tables that are defined by <temporary table declaration>s that are contained in the <SQL-client module definition> of *p*, and the cursor instance descriptors of all open cursors that are not global extended dynamic cursors and whose SQL-client module is the SQL-client module of *p* are removed from *RSC*.

f) After the completion of *p*, it is implementation-defined whether open cursors declared in the <SQL-client module definition> of *p* are closed and destroyed, whether local temporary tables associated with *RCS* are destroyed, and whether prepared statements prepared by *p* are deallocated.

g) If *R* is an SQL-invoked procedure, then for each SQL parameter that is an output SQL parameter or both an input and output SQL parameter whose corresponding argument was not assigned a value, that corresponding argument is set to an implementation-defined value of the appropriate type.

h) If the external security characteristic of an external SQL-invoked routine is IMPLEMENTATION DEFINED, then the user identifier and role name in the first cell of the authorization stack of the new SQL-session context are implementation-defined.

52) Subclause 10.9, "<aggregate function>":

a) If COUNT is specified, then the declared type of the result is an implementation-defined exact numeric type with scale of 0 (zero).

b) If SUM or AVG is specified, then:

i) If SUM is specified and the declared type of the argument is exact numeric with scale *S*, then the declared type of the result is an implementation-defined exact numeric type with scale *S*.

ii) If AVG is specified and the declared type *DT* of the argument is exact numeric, then the declared type of the result is an implementation-defined exact numeric type with precision not less than the precision of *DT* and scale not less than the scale of *DT*.

iii) If the declared type *DT* of the argument is approximate numeric, then the declared type of the result is an implementation-defined approximate numeric type with precision not less than the precision of *DT*.

c) If VAR_POP or VAR_SAMP is specified, then the declared type of the result is an implementation-defined approximate numeric type. If the declared type of the argument is approximate numeric, then the precision of the result is not less than the precision of the argument.

d) If <binary set function type> is specified, then

Case:

i) If REGR_COUNT is specified, then the declared type of the result is an implementation-defined exact numeric type with scale of 0 (zero).

ii) Otherwise, the declared type of the result is an implementation-defined approximate numeric type. If either argument is approximate numeric, then the precision of the result shall be at least as great as the precision of the approximate numeric argument(s).

e) If <hypothetical set function> is specified, then

Case:

i) If RANK or DENSE_RANK is specified, then the declared type of the result is exact numeric with implementation-defined precision and with scale 0 (zero).

ii) Otherwise, the declared type of the result is approximate numeric with implemenation-defined precision.

f) If the declared type of the <value expression> simply contained in the <sort specification> of an <inverse distribution function> that specifies PERCENTILE_CONT is numeric, then the result type is approximate numeric with implementation-defined precision.

53) Subclause 10.10, "<sort specification list>":

a) If <null ordering> is not specified, then an implementation-defined <null ordering> is implicit. The implementation-defined default for <null ordering> shall not depend on the context outside of <sort specification list>.

54) Subclause 11.1, "<schema definition>":

a) If <schema character set specification> is not specified, then a <schema character set specification> containing an implementation-defined <character set specification> is implicit.

b) If <schema path specification> is not specified, then a <schema path specification> containing an implementation-defined <schema name list> is implicit.

c) If AUTHORIZATION <authorization identifier> is not specified, then an <authorization identifier> equivalent to the implementation-defined <authorization identifier> for the SQL-session is implicit.

d) The privileges necessary to execute the <schema definition> are implementation-defined.

55) Subclause 11.6, "<table constraint definition>":

a) The ordering of the lists of referencing column names and referenced column names in a referential constraint descriptor is implementation-defined, but shall be such that corresponding column names occupy corresponding positions in each list.

56) Subclause 11.33, "<collation definition>":

a) The <existing collation name>s that are supported are implementation-defined.

b) The collation resulting from the specification of EXTERNAL in a <collation definition> may be implementation-defined.

57) Subclause 11.35, "<transliteration definition>":

a) The <existing transliteration name>s that are supported are implementation-defined.

58) Subclause 11.39, "<trigger definition>":

a) It is implementation-defined whether the <triggered SQL statement> shall not generally contain an <SQL transaction statement>, an <SQL connection statement>, an <SQL schema statement>, an <SQL dynamic statement>, or an <SQL session statement>.

b) It is implementation-defined whether the <triggered action> shall not generally contain an <SQL data change statement> or a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

59) Subclause 11.50, "<SQL-invoked routine>":

a) If READS SQL DATA is specified, then it is implementation-defined whether the SQL routine body shall not contain an <SQL procedure statement> *S* that satisfies at least one of the following:

i) *S* is an <SQL data change statement>.

ii) *S* contains a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

iii) *S* contains an <SQL procedure statement> that is an <SQL data change statement>.

b) If CONTAINS SQL is specified, then it is implementation-defined whether the SQL routine body shall not contain an <SQL procedure statement> *S* that satisfies at least one of the following:

i) *S* is an <SQL data statement> other than <free locator statement> and <hold locator statement>.

ii) *S* contains a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data or possibly reads SQL-data.

iii) *S* contains an <SQL procedure statement> that is an <SQL data statement> other than <free locator statement> and <hold locator statement>.

c) If DETERMINISTIC is specified, then it is implementation-defined whether the <SQL routine body> shall not contain an <SQL procedure statement> that is possibly non-deterministic.

d) It is implementation-defined whether the <SQL routine body> shall not contain an <SQL connection statement>, an <SQL schema statement>, an <SQL dynamic statement>, or an <SQL transaction statement> other than a <savepoint statement>, <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>.

60) Subclause 11.62, "<sequence generator definition>":

a) If <sequence generator data type option> is not specified, then an implementation-defined exact numeric type *DT* with scale 0 (zero) is implicit.

61) Subclause 12.4, "<role definition>":

a) The Access Rules are implementation-defined.

62) Subclause 12.7, "<revoke statement>":

a) When loss of the USAGE privilege on a character set causes an SQL-client module to be determined to be a lost module, the impact on that SQL-client module is implementation-defined.

63) Subclause 13.1, "<SQL-client module definition>":

a) If the explicit or implicit <schema name> does not specify a <catalog name>, then an implementation-defined <catalog name> is implicit.

b) If <module path specification> is not specified, then a <module path specification> containing an implementation-defined <schema name list> is implicit.

c) If a <module character set specification> is not specified, then a <module character set specification> that specifies the implementation-defined character set that contains every character that is in <SQL language character> is implicit.

64) Subclause 14.4, "<open statement>":

a) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

65) Subclause 14.9, "<delete statement: searched>":

a) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

b) If there is any sensitive cursor *CR* that is open in the SQL-transaction in which this statement is being executed and *CR* has been held into a subsequent SQL-transaction, then whether the change resulting from the successful execution of this statement is made visible to *CR* is implementation-defined.

66) Subclause 14.10, "<insert statement>":

a) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

b) If there is any sensitive cursor *CR* that is open in the SQL-transaction in which this statement is being executed and *CR* has been held into a subsequent SQL-transaction, then whether the change resulting from the successful execution of this statement is made visible to *CR* is implementation-defined.

67) Subclause 14.11, "<merge statement>":

a) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

b) If there is any sensitive cursor *CR* that is open in the SQL-transaction in which this statement is being executed and *CR* has been held into a subsequent SQL-transaction, then whether the change resulting from the successful execution of this statement is made visible to *CR* is implementation-defined.

68) Subclause 14.13, "<update statement: searched>":

a) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

b) If there is any sensitive cursor *CR* that is open in the SQL-transaction in which this statement is being executed and *CR* has been held into a subsequent SQL-transaction, then whether the change resulting from the successful execution of this statement is made visible to *CR* is implementation-defined.

69) Subclause 15.1, "Effect of opening a cursor":

a) Whether an SQL-implementation is able to disallow significant changes that would not be visible through a currently open cursor is implementation-defined.

70) Subclause 15.5, "Effect of a positioned delete":

a) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

b) If there is any sensitive cursor *CR* that is open in the SQL-transaction in which this statement is being executed and *CR* has been held into a subsequent SQL-transaction, then whether the change resulting from the successful execution of this statement is made visible to *CR* is implementation-defined.

c) Whether the current row is removed from the sequence of rows of the result set descriptor of *CR* is implementation-defined.

71) Subclause 15.6, "Effect of a positioned update":

a) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined.

b) If there is any sensitive cursor *CR* that is open in the SQL-transaction in which this statement is being executed and *CR* has been held into a subsequent SQL-transaction, then whether the change resulting from the successful execution of this statement is made visible to *CR* is implementation-defined.

72) Subclause 17.2, "<set transaction statement>":

a) The isolation level that is set for a transaction is an implementation-defined isolation level that will not exhibit any of the phenomena that the explicit or implicit <level of isolation> would not exhibit, as specified in Table 8, "SQL-transaction isolation levels and the three phenomena".

73) Subclause 17.5, "<savepoint statement>":

a) The maximum number of savepoints per SQL-transaction is implementation-defined.

74) Subclause 17.7, "<commit statement>":

a) Whether a prepared statement, other than the prepared statement of a held cursor, remains in existence is implementation-defined.

75) Subclause 17.8, "<rollback statement>":

a) The status of any open cursors in any SQL-client module associated with the current SQL-transaction that were opened by that SQL-transaction before the establishment of a savepoint to which a rollback is executed is implementation-defined.

76) Subclause 18.1, "<connect statement>":

a) If <connection user name> is not specified, then an implementation-defined <connection user name> for the SQL-connection is implicit.

b) The initial value of the current role name is the null value.

c) The restrictions on whether or not the <connection user name> shall be identical to the <module authorization identifier> for the SQL-client module that contains the <externally-invoked procedure> that contains the <connect statement> are implementation-defined.

d) If DEFAULT is specified, then the method by which the default SQL-server is determined is implementation-defined.

e) The method by which <SQL-server name> is used to determine the appropriate SQL-server is implementation-defined.

77) Subclause 19.2, "<set session user identifier statement>":

a) Implementation-defined restrictions on the user identifiers to which the SQL-session user identifier can be set are permitted, other than to disallow setting it to its current value.

78) Subclause 19.10, "<set session collation statement>":

a) If no <character set specification> is specified in a <set session collation statement>, then the character sets for which the SQL-session collations are set are implementation-defined.

79) Subclause 20.2, "<allocate descriptor statement>":

a) If WITH MAX <occurrences> is not specified, then an implementation-defined default value for <occurrences> that is greater than 0 (zero) is implicit.

b) The maximum number of SQL descriptor areas and the maximum number of item descriptor areas for a single SQL descriptor area are implementation-defined.

80) Subclause 20.5, "<set descriptor statement>":

a) Restrictions on changing TYPE, LENGTH, OCTET_LENGTH, SCALE, COLLATION_CATALOG, COLLATION_SCHEMA, COLLATION_NAME, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_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.

81) Subclause 20.6, "<prepare statement>":

a) The Format and Syntax Rules for a <preparable implementation-defined statement> are implementation-defined.

82) Subclause 20.9, "<describe statement>":

a) The character set of the data type of <descriptor name> is implementation-defined.

b) If *SR* does not contain a single routine SQL-invoked *R*, then the values of PARAMETER_MODE, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, and PARAMETER_SPECIFIC_NAME in the descriptor for each <dynamic parameter specification> simply contained in the <call statement> are set to implementation-defined values.

83) Subclause 21.1, "<embedded SQL host program>":

a) If *H* does not contain an <embedded authorization declaration> that specifies SCHEMA, then the <schema name> of the <module authorization clause> of *m* is implementation-defined.

b) If *H* does not contain an <embedded authorization declaration>, then *m* contains a <module authorization clause> that specifies "SCHEMA *SN*", where *SN* is an implementation-defined <schema name>.

c) If an <embedded character set declaration> is not specified, then an <embedded character set declaration> containing an implementation-defined <character set specification> is implicit.

d) Each <allocate cursor statement> is replaced with a host language procedure or subroutine call of an implementation-defined procedure that associates the <dynamic cursor name> with the prepared statement.

e) If an <embedded SQL host program> does not contain an <embedded path specification>, then the implied module contains an implementation-defined <module path specification>.

84) Subclause 21.4, "<embedded SQL C program>":

a) The implicit character set in a <C character variable>, a <C VARCHAR variable>, or a <C CLOB variable>is implementation-defined.

85) Subclause 21.5, “<embedded SQL COBOL program>”:

a) The COBOL data description clauses, in addition to the PICTURE, SIGN, USAGE and VALUE clauses, that may appear in a <COBOL variable definition> are implementation-defined.

86) Subclause 21.9, “<embedded SQL PL/I program>”:

a) The PL/I data description clauses, in addition to the <PL/I type specification> and the INITIAL clause, that may appear in a <PL/I variable definition> are implementation-defined.

87) Subclause 22.1, “<direct SQL statement>”:

a) The <value specification> that represents the null value is implementation-defined.

b) The Format, Syntax Rules, and Access Rules for <direct implementation-defined statement> are implementation-defined.

c) Whether a <direct implementation-defined statement> may be associated with an active transaction is implementation-defined.

d) Whether a <direct implementation-defined statement> initiates a transaction is implementation-defined.

88) Subclause 23.1, “<get diagnostics statement>”:

a) The maximum length of each variable-length character item in the diagnostics area is in most cases implementation-defined.

b) The character string value set for CLASS_ORIGIN and SUBCLASS_ORIGIN for an implementation-defined class code or subclass code is implementation-defined, but shall not be 'ISO 9075'.

c) The value of MESSAGE_TEXT is an implementation-defined character string.

d) Negative values of COMMAND_FUNCTION_CODE are implementation-defined and indicate implementation-defined SQL-statements.

e) The order in which the occupied condition area numbers are presented when ALL is specified without a condition number is implementation-defined.

89) Subclause 24.1, “SQLSTATE”:

a) The character set associated with the class value and subclass value of the SQLSTATE parameter is implementation-defined.

b) The values and meanings for classes and subclasses 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 implementation-defined. The values and meanings for all subclasses that are associated with implementation-defined class values are implementation-defined.

90) Clause 25, “Conformance”:

a) The method of flagging nonconforming SQL language or processing of conforming SQL language is implementation-defined, as is the list of additional <key word>s that may be required by the SQL-implementation.

(Blank page)

Annex C
(informative)

Implementation-dependent elements

This Annex is modified by Annex D, "Implementation-dependent elements", in ISO/IEC 9075-3.
This Annex is modified by Annex C, "Implementation-dependent elements", in ISO/IEC 9075-4.
This Annex is modified by Annex C, "Implementation-dependent elements", in ISO/IEC 9075-9.
This Annex is modified by Annex C, "Implementation-dependent elements", in ISO/IEC 9075-10.
This Annex is modified by Annex C, "Implementation-dependent elements", in ISO/IEC 9075-13.
This Annex is modified by Annex C, "Implementation-dependent elements", in ISO/IEC 9075-14.

This Annex references those places where this part of ISO/IEC 9075 states explicitly that the actions of a conforming SQL-implementation are implementation-dependent.

1) Subclause 3.1.6, "Definitions provided in Part 2":

   a) Whether two non-null values of user-defined type whose comprison form is RELATIVE or MAP that result in *Unknown* when tested for equality according to the rules of Subclause 8.2, "<comparison predicate>", are distinct or not is implementation-dependent.

2) Subclause 4.1, "Data types":

   a) The physical representation of a value of a data type is implementation-dependent.

   b) The null value or values for each data type is implementation-dependent.

3) Subclause 4.14, "Tables":

   a) Because global temporary table contents are distinct within SQL-sessions, and created local temporary tables are distinct within SQL-client modules within SQL-sessions, the effective <schema name> of the schema in which the global temporary table or the created local temporary table is instantiated is an implementation-dependent <schema name> that may be thought of as having been effectively derived from the <schema name> of the schema in which the global temporary table or created local temporary table is defined and the implementation-dependent SQL-session identifier associated with the SQL-session.

   b) The effective <schema name> of the schema in which the created local temporary table is instantiated may be thought of as being further qualified by a unique implementation-dependent name associated with the SQL-client module in which the created local temporary table is referenced.

4) Subclause 4.14.9, "Windowed tables":

   a) The window name of a window defined implicitly by an <in-line window specification> is implementation-dependent.

5) Subclause 4.27.5, "Result sets returned by SQL-invoked procedures":

   a) It is implementation-dependent whether or not result sets whose positions are greater than that maximum number are returned.

6) Subclause 4.30, “Diagnostics area”:

   a) The condition area limit is implementation-dependent when not explicitly specified.

   b) The ordering of the information about conditions placed into the diagnostics area is implementation-dependent, except that the first condition in the diagnostics area always corresponds to the condition corresponding to the SQLSTATE value.

   c) The maximum number of diagnostics area in a diagnostics area stack is implementation-defined.

7) Subclause 4.32, “Cursors”:

   a) If the <cursor specification> does not contain an <order by clause>, or contains an <order by clause> that does not specify the order of the rows completely, then the rows of the result set have an order that is defined only to the extent that the <order by clause> specifies an order and is otherwise implementation-dependent.

   b) The effect on the position and state of an open cursor when an error occurs during the execution of an SQL-statement that identifies the cursor is implementation-dependent.

   c) If an asensitive cursor is open and a change is made to SQL-data from within the same SQL-transaction other than through that cursor, then whether that change will be visible through that cursor before it is closed is implementation-dependent.

8) Subclause 4.35, “SQL-transactions”:

   a) The schema definitions that are implicitly read on behalf of executing an SQL-statement are implementation-dependent.

9) Subclause 4.37, “SQL-sessions”:

   a) A unique implementation-dependent SQL-session identifier is associated with each SQL-session.

10) Subclause 4.39, “Client-server operation”:

   a) The <SQL-client module name> of the SQL-client module that is effectively materialized on an SQL-server is implementation-dependent.

   b) Following the execution of an <SQL procedure statement> by an SQL-server, diagnostic information is passed in an implementation-dependent manner into the SQL-agent's diagnostics area stack in the SQL-client.

   c) The effect on diagnostic information of incompatibilities between the character repertoires supported by the SQL-client and SQL-server environments is implementation-dependent.

11) Subclause 6.7, “<column reference>”:

   a) If *QCR* is a group-invariant column reference and the most specific type of *QCR* is character string, datetime with time zone, or a user-defined type, then *QCR* denotes an implementation-dependent value that is not distinct from the value of *c* in every row of a given group of the qualifying query of *QCR*.

12) Subclause 6.9, “<set function specification>”:

   a) The maximum or minimum of a set of values of a user-defined type is implementation-dependent if the comparison of at least two values of the set results in *Unknown*.

13) Subclause 6.12, “<cast specification>”:

a) When a multiset is cast to an array type, the order of elements in the result is implementation-dependent.

14) Subclause 6.27, “<numeric value function>”:

a) If <regex occurrences function> specifies OCTETS and the octet whose position is given by the <start position> is not the first octet of a character, then the result of <regex occurrences function> is implementation-dependent.

b) If <regex position expression> specifies OCTETS and the octet whose position is given by the <start position> is not the first octet of a character, then the result of <regex position expression> is implementation-dependent.

15) Subclause 6.29, “<string value function>”:

a) If <regex substring function> specifies OCTETS and the octet whose position is given by the <start position> is not the first octet of a character, then the result of <regex occurrences function> is implementation-dependent.

b) If <regex transliteration> specifies OCTETS and the octet whose position is given by the <start position> is not the first octet of a character, then the result of <regex position expression> is implementation-dependent.

16) Subclause 6.31, “<datetime value function>”:

a) The time of evaluation of the CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP functions during the execution of an SQL-statement is implementation-dependent.

17) Subclause 6.32, “<interval value expression>”:

a) The start datetime used for converting intervals to scalars for subtraction purposes is implementation-dependent.

18) Subclause 6.36, “<array value constructor>”:

a) The order of array elements in the result of an <array value constructor by query> which is not decided by the <order by clause> is implementation-dependent.

19) Subclause 7.1, “<row value constructor>”:

a) The names of the fields of a <row value constructor> that specifies a <row value constructor element list> are implementation-dependent.

b) The names of the fields of a <contextually typed row value constructor> are implementation-dependent.

20) Subclause 7.3, “<table value constructor>”:

a) The column names of a <table value constructor> or a <contextually typed table value constructor> are implementation-dependent.

21) Subclause 7.9, “<group by clause>”:

a) If the declared type of a grouping column is a user-defined type and the comparison of that column for two rows results in *Unknown*, then the assignment of those rows to groups in the result of the <group by clause> is implementation-dependent.

b) When a <search condition> or <value expression> is applied to a group, the value of a group-invariant column reference whose most specific type is character string, datetime with time zone or a user-defined

type, and that references a column that is functionally dependent on the grouping columns is implementation-dependent.

22) Subclause 7.11, "<window clause>":

a) If the window ordering clause of a window structure descriptor is absent, then the window ordering is implementation-dependent.

b) The window ordering of peer rows within a window partition is implementation-dependent, but the window ordering shall be the same for all window structure descriptors that are order-equivalent. It shall also be the same for windows *W1* and *W2* if *W1* is the ordering window for *W2*.

23) Subclause 7.12, "<query specification>":

a) When a column is not named by an <as clause> and is not derived from a single column reference, then the name of the column is implementation-dependent.

b) If the <set quantifier> DISTINCT is specified, and the most specific type of a result column is character string, datetime with time zone or a user-defined type, then the precise values retained in that column after eliminating redundant duplicates is implementation-dependent.

24) Subclause 7.13, "<query expression>":

a) If a <simple table> is neither a <query specification> nor an <explicit table>, then the name of each column of the <simple table> is implementation-dependent.

b) If a <query term> immediately contains INTERSECT and the <column name>s of a pair of corresponding columns of the operand tables are not equivalent, then the result column has an implementation-dependent <column name>.

c) If a <query expression body> immediately contains UNION or INTERSECT, and the <column name>s of a pair of corresponding columns of the operand tables are not equivalent, then the result column has an implementation-dependent <column name>.

25) Subclause 8.2, "<comparison predicate>":

a) When the operations MAX, MIN, DISTINCT, and references to a grouping column refer to a variable-length character string, the specific value selected from the set of equal values is implementation-dependent.

26) Subclause 10.4, "<routine invocation>":

a) Each SQL argument A_i in *SAL* is evaluated, in an implementation-dependent order, to obtain a value V_i.

27) Subclause 10.9, "<aggregate function>":

a) If the declared type of the argument of MAX or MIN is a user-defined type and the comparison of two values results in *Unknown*, then the maximum or minimum is implementation-dependent.

28) Subclause 10.10, "<sort specification list>":

a) If PV_i and QV_i are not the null value and the result of "PV_i <comp op> QV_i" is *Unknown*, then the relative ordering of PV_i and QV_i is implementation-dependent.

b) The relative ordering of two rows that are not distinct with respect to the <sort specification> is implementation-dependent.

29) Subclause 11.6, “<table constraint definition>”:

a) The <constraint name> of a constraint that does not specify a <constraint name definition> is implementation-dependent.

30) Subclause 11.8, “<referential constraint definition>”:

a) The specific value to use for cascading among various values that are not distinct is implementation-dependent.

31) Subclause 11.24, “<domain definition>”:

a) The <constraint name> of a constraint that does not specify a <constraint name definition> is implementation-dependent.

32) Subclause 11.33, “<collation definition>”:

a) The collation of characters for which a collation is not otherwise specified is implementation-dependent.

33) Subclause 14.5, “<fetch statement>”:

a) The order of assignment to targets in the <fetch target list> of values returned by a <fetch statement>, other than status parameters, is implementation-dependent.

b) If an error occurs during assignment of a value to a target during the execution of a <select statement: single row>, then the values of targets other than status parameters are implementation-dependent.

c) If an exception condition occurs during the assignment of a value to a target, then the values of all targets are implementation-dependent and *CR* remains positioned on the current row.

d) It is implementation-dependent whether *CR* remains positioned on the current row when an exception condition is raised during the derivation of any <derived column>.

34) Subclause 14.7, “<select statement: single row>”:

a) The order of assignment to targets in the <select target list> of values returned by a <select statement: single row>, other than status parameters, is implementation-dependent.

b) If the cardinality of the <query expression> is greater than 1 (one), then it is implementation-dependent whether or not values are assigned to the targets identified by the <select target list>.

c) If an error occurs during assignment of a value to a target during the execution of a <select statement: single row>, then the values of targets other than status parameters are implementation-dependent.

35) Subclause 14.10, “<insert statement>”:

a) The generation of the value of a derived self-referencing column is implementation-dependent.

36) Subclause 14.11, “<merge statement>”:

a) The generation of the value of a derived self-referencing column is implementation-dependent.

37) Subclause 15.1, “Effect of opening a cursor”:

a) If a <cursor specification> does not contain an <order by clause>, then the ordering of rows in the result set associated with that <cursor specification> is implementation-dependent.

b) If a <cursor specification> contains an <order by clause> and a group of two or more rows in the result set associated with that <declare cursor> contain values that

Case:

i) are the null value, or

ii) compare equal according to Subclause 8.2, "<comparison predicate>"

in all columns specified in the <order by clause>, then the order in which rows in that group are returned is implementation-dependent.

c) The relative ordering of two non-null values of a user-defined type *UDT* whose comparison as determined by the user-defined ordering of *UDT* is *Unknown* is implementation-dependent.

38) Subclause 17.2, "<set transaction statement>":

a) If <number of conditions> is not specified, then an implementation-dependent value not less than 1 (one) is implicit.

39) Subclause 18.3, "<disconnect statement>":

a) If ALL is specified, then *l* is a list representing every active SQL-connection that has been established by a <connect statement> by the current SQL-agent and that has not yet been disconnected by a <disconnect statement>, in an implementation-dependent order.

40) Subclause 20.4, "<get descriptor statement>":

a) 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.

b) For a <dynamic parameter specification>, the value of UNNAMED is 1 (one) and the value of NAME is implementation-dependent.

c) The value retrieved by a <get descriptor statement> for any field whose value is undefined is implementation-dependent.

41) Subclause 20.5, "<set descriptor statement>":

a) If an exception condition is raised in a <set descriptor statement>, then the values of all elements of the descriptor specified in the <set descriptor statement> are implementation-dependent.

42) Subclause 20.10, "<input using clause>":

a) When a <describe input statement> is used, the values for NAME, DATA, and INDICATOR in the SQL dynamic descriptor area structure is implementation-dependent. If TYPE indicates a character string type or a binary large object type, then the values of SCALE and PRECISION are implementation-dependent. If TYPE indicates an exact or approximate numeric type, then the values of LENGTH and OCTET_LENGTH are implementation-dependent. If TYPE indicates a boolean type, then the values of PRECISION, SCALE, LENGTH, and OCTET_LENGTH are implementation-dependent.

43) Subclause 20.11, "<output using clause>":

a) When a <describe output statement> is executed, the values of DATA and INDICATOR are implementation-dependent. If TYPE indicates a character string type or a binary large object type, then the values of SCALE and PRECISION are implementation-dependent. If TYPE indicates an exact or approximate numeric type, then the values of LENGTH and OCTET_LENGTH are implementation-dependent. If TYPE indicates a boolean type, then the values of PRECISION, SCALE, LENGTH, and OCTET_LENGTH are implementation-dependent.

44) Subclause 20.13, "<execute immediate statement>":

a) The <statement name> of the statement prepared when an <execute immediate statement> is executed is implementation-dependent.

45) Subclause 21.1, "<embedded SQL host program>":

a) The <SQL-client module name> of the implied <SQL-client module definition> derived from an <embedded SQL host program> is implementation-dependent.

b) If an <embedded SQL host program> does not contain an <embedded authorization declaration>, then the <module authorization identifier> of the implied <SQL-client module definition> derived from the <embedded SQL host program> is implementation-dependent.

c) In each <declare cursor> in the implied <SQL-client module definition> derived from an <embedded SQL host program>, each <embedded variable name> has been replaced consistently with a distinct <host parameter name> that is implementation-dependent.

d) The <procedure name> of each <externally-invoked procedure> in the implied <SQL-client module definition> derived from an <embedded SQL host program> is implementation-dependent.

e) In each <externally-invoked procedure> in the implied <SQL-client module definition> derived from an <embedded SQL host program>, each <embedded variable name> has been replaced consistently with a distinct <host parameter name> that is implementation-dependent.

f) For <SQL procedure statement>s other than <open statement>s, whether one <externally-invoked procedure> in the implied <SQL-client module definition> derived from an <embedded SQL host program> can correspond to more than one <SQL procedure statement> in the <embedded SQL host program> is implementation-dependent.

g) In each <externally-invoked procedure> in the implied <SQL-client module definition> derived from an <embedded SQL host program>, the order of the instances of <host parameter declaration> is implementation-dependent.

46) Subclause 22.1, "<direct SQL statement>":

a) A <commit statement> or a <rollback statement> is 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.

47) Subclause 23.1, "<get diagnostics statement>":

a) 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.

b) If <condition number> has a value other than 1 (one), then the association between <condition number> values and specific conditions raised during evaluation of the General Rules for that SQL-statement is implementation-dependent.

c) The order in which the statement information items and the condition information items of a particular condition area are presented when ALL is specified is implementation-dependent.

(Blank page)

Annex D
(informative)

Deprecated features

This Annex is modified by Annex C, "Deprecated features", in ISO/IEC 9075-11.

It is intended that the following features will be removed at a later date from a revised version of this part of ISO/IEC 9075:

None.

(Blank page)

Annex E
(informative)

Incompatibilities with ISO/IEC 9075-2:2003

This Annex is modified by Annex E, "Incompatibilities with ISO/IEC 9075:2003", in ISO/IEC 9075-3.
This Annex is modified by Annex D, "Incompatibilities with ISO/IEC 9075:1999", in ISO/IEC 9075-4.
This Annex is modified by Annex D, "Incompatibilities with ISO/IEC 9075:1999", in ISO/IEC 9075-9.
This Annex is modified by Annex D, "Incompatibilities with ISO/IEC 9075:2003", in ISO/IEC 9075-11.
This Annex is modified by Annex D, "Incompatibilities with ISO/IEC 9075:2003", in ISO/IEC 9075-14.

This edition of this part of ISO/IEC 9075 introduces some incompatibilities with the earlier version of Database Language SQL as specified in ISO/IEC 9075-2:2003.

Except as specified in this Annex, features and capabilities of Database Language SQL are compatible with ISO/IEC 9075-2:2003.

1) In ISO/IEC 9075:2-2003, <search or cycle clause>'s <recursive search order> required the specification of <sort specification list>. This edition of ISO/IEC 9075-2 further restricts that specification to <column name list>.

2) In ISO/IEC 9075-2:2003, the declared type of a <binary string literal> was binary large object string type. In this edition of ISO/IEC 9075-2, it is implementation-defined whether the declared type of a <binary string literal> is binary large object string type, variable-length binary string, or fixed-length binary string.

3) In ISO/IEC 9075-2:2003, the <non-reserved word> EXCEPTION could be used as a synonym for the <non-reserved word> CONDITION in a <get diagnostics statement>. In this edition of ISO/IEC 9075-2, the <non-reserved word> EXCEPTION has been omitted entirely.

4) A number of additional <reserved word>s have been added to the language. These <reserved word>s are:

   — CURRENT_CATALOG

   — CURRENT_SCHEMA

   — LIKE_REGEX

   — OCCURRENCES_REGEX

   — POSITION_REGEX

   — SUBSTRING_REGEX

   — TRANSLATE_REGEX

   — VARBINARY

(Blank page)

Annex F
(informative)

SQL feature taxonomy

This Annex describes a taxonomy of features and packages defined in this part of ISO/IEC 9075.

Table 36, “Feature taxonomy and definition for mandatory features”, contains a taxonomy of the mandatory features of SQL language that are specified in this part of ISO/IEC 9075. Table 37, “Feature taxonomy for optional features”, contains a taxonomy of the optional features of the SQL language that are specified in this part of ISO/IEC 9075.

In these tables, the first column contains a counter that may be used to quickly locate rows of the table; these values otherwise have no use and are not stable — that is, they are subject to change in future editions of or even Technical Corrigenda to ISO/IEC 9075 without notice.

The “Feature ID” column of Table 36, “Feature taxonomy and definition for mandatory features”, and of Table 37, “Feature taxonomy for optional features”, specifies the formal identification of each feature and each subfeature contained in the table.

The “Feature Name” column of Table 36, “Feature taxonomy and definition for mandatory features”, contains a brief description of the feature or subfeature associated with the Feature ID value.

The “Feature Description” column of Table 36, “Feature taxonomy and definition for mandatory features”, provides the only definition of the mandatory features of this part of ISO/IEC 9075. This definition consists of indications of specific language elements supported in each feature, subject to the constraints of all Syntax Rules, Access Rules, and Conformance Rules.

Table 36 — Feature taxonomy and definition for mandatory features

	Feature ID	Feature Name	Feature Description
1	**E011**	**Numeric data types**	Subclause 6.1, “<data type>”, <numeric type>, including numeric expressions, numeric literals, numeric comparisons, and numeric assignments
2	E011-01	INTEGER and SMALLINT data types (including all spellings)	— Subclause 5.2, “<token> and <separator>”: The <reserved word>s INT, INTEGER, and SMALLINT — Subclause 5.3, “<literal>”: `[<sign>] <unsigned integer>` — Subclause 6.1, “<data type>”: The INTEGER and SMALLINT <exact numeric type>s — Subclause 13.6, “Data type correspondences”: Type correspondences for INTEGER and SMALLINT for all supported languages

	Feature ID	Feature Name	Feature Description
3	E011-02	REAL, DOUBLE PRECISON, and FLOAT data types	— Subclause 5.2, “<token> and <separator>”: The <reserved word>s REAL, DOUBLE, FLOAT, and PRECISION — Subclause 5.3, “<literal>”: `[<sign>] <approximate numeric literal>` — Subclause 6.1, “<data type>”: <approximate numeric type> — Subclause 13.6, “Data type correspondences”: Type correspondences for REAL, DOUBLE PRECISION, and FLOAT for all supported languages
4	E011-03	DECIMAL and NUMERIC data types	— Subclause 5.2, “<token> and <separator>”: The <reserved word>s DEC, DECIMAL, and NUMERIC — Subclause 5.3, “<literal>”: `[<sign>] <exact numeric literal>` — Subclause 6.1, “<data type>”: The DECIMAL and NUMERIC <exact numeric type>s — Subclause 13.6, “Data type correspondences”: Type correspondences for DECIMAL and NUMERIC for all supported languages
5	E011-04	Arithmetic operators	— Subclause 6.26, “<numeric value expression>”: When the <numeric primary> is a <value expression primary>
6	E011-05	Numeric comparison	— Subclause 8.2, “<comparison predicate>”: For the numeric data types, without support for <table subquery> and without support for Feature F131, “Grouped operations”
7	E011-06	Implicit casting among the numeric data types	— Subclause 8.2, “<comparison predicate>”: Values of any of the numeric data types can be compared to each other; such values are compared with respect to their algebraic values — Subclause 9.1, “Retrieval assignment”, and Subclause 9.2, “Store assignment”: Values of one numeric type can be assigned to another numeric type, subject to rounding, truncation, and out of range conditions
8	**E021**	**Character string types**	— Subclause 6.1, “<data type>”: <character string type>, including character expressions, character literals, character comparisons, character assignments, and other operations on character data

	Feature ID	Feature Name	Feature Description
9	E021-01	CHARACTER data type (including all its spellings)	— Subclause 5.2, “<token> and <separator>”: The <reserved word>s CHAR and CHARACTER — Subclause 6.1, “<data type>”: The CHARACTER <character string type> — Subclause 6.28, “<string value expression>”: For values of type CHARACTER — Subclause 13.6, “Data type correspondences”: Type correspondences for CHARACTER for all supported languages
10	E021-02	CHARACTER VARYING data type (including all its spellings)	— Subclause 5.2, “<token> and <separator>”: The <reserved word>s VARCHAR and VARYING — Subclause 6.1, “<data type>”: The CHARACTER VARYING <character string type> — Subclause 6.28, “<string value expression>”: For values of type CHARACTER VARYING — Subclause 13.6, “Data type correspondences”: Type correspondences for CHARACTER VARYING for all supported languages
11	E021-03	Character literals	— Subclause 5.3, “<literal>”: `<quote> [ <character representation>... ] <quote>`
12	E021-04	CHARACTER_LENGTH function	— Subclause 6.27, “<numeric value function>”: The <char length expression>
13	E021-05	OCTET_LENGTH function	— Subclause 6.27, “<numeric value function>”: The <octet length expression>
14	E021-06	SUBSTRING function	— Subclause 6.29, “<string value function>”: The <character substring function>
15	E021-07	Character concatenation	— Subclause 6.28, “<string value expression>”: The <concatenation> expression
16	E021-08	UPPER and LOWER functions	— Subclause 6.29, “<string value function>”: The <fold> function
17	E021-09	TRIM function	— Subclause 6.29, “<string value function>”: The <trim function>
18	E021-10	Implicit casting among the fixed-length and variable-length character string types	— Subclause 8.2, “<comparison predicate>”: Values of either the CHARACTER or CHARACTER VARYING data types can be compared to each other — Subclause 9.1, “Retrieval assignment”, and Subclause 9.2, “Store assignment”: Values of either the CHARACTER or CHARACTER VARYING data type can be assigned to the other type, subject to truncation conditions

	Feature ID	Feature Name	Feature Description
19	E021-11	POSITION function	— Subclause 6.27, “<numeric value function>”: The <position expression>
20	E021-12	Character comparison	— Subclause 8.2, “<comparison predicate>”: For the CHARACTER and CHARACTER VARYING data types, without support for <table subquery> and without support for Feature F131, “Grouped operations”
21	**E031**	**Identifiers**	— Subclause 5.2, “<token> and <separator>”: <regular identifier> and <delimited identifier>
22	E031-01	Delimited identifiers	— Subclause 5.2, “<token> and <separator>”: <delimited identifier>
23	E031-02	Lower case identifiers	— Subclause 5.2, “<token> and <separator>”: An alphabetic character in a <regular identifier> can be either lower case or upper case (meaning that non-delimited identifiers need not comprise only upper case letters)
24	E031-03	Trailing underscore	— Subclause 5.2, “<token> and <separator>”: The list <identifier part> in a <regular identifier> can be an <underscore>
25	**E051**	**Basic query specification**	— Subclause 7.12, “<query specification>”: When <table reference> is a <table or query name> that is a <table name>, without the support of Feature F131, “Grouped operations”
26	E051-01	SELECT DISTINCT	— Subclause 7.12, “<query specification>”: With a <set quantifier> of DISTINCT, but without subfeatures E051-02 through E051-09
27	E051-02	GROUP BY clause	— Subclause 7.4, “<table expression>”: <group by clause>, but without subfeatures E051-04 through E051-09 — Subclause 7.9, “<group by clause>”: With the restrictions that the <group by clause> shall contain all non-aggregated columns in the <select list> and that any column in the <group by clause> shall also appear in the <select list>
28	E051-04	GROUP BY can contain columns not in <select list>	— Subclause 7.9, “<group by clause>”: Without the restriction that any column in the <group by clause> shall also appear in the <select list>
29	E051-05	Select list items can be renamed	— Subclause 7.12, “<query specification>”: <as clause>

	Feature ID	Feature Name	Feature Description
30	E051-06	HAVING clause	— Subclause 7.4, "<table expression>": <having clause> — Subclause 7.10, "<having clause>"
31	E051-07	Qualified * in select list	— Subclause 7.12, "<query specification>": <qualified asterisk>
32	E051-08	Correlation names in the FROM clause	— Subclause 7.6, "<table reference>": `[ AS ] <correlation name>`
33	E051-09	Rename columns in the FROM clause	— Subclause 7.6, "<table reference>": `[ AS ] <correlation name> [ <left paren> <derived column list> <right paren> ]`
34	**E061**	**Basic predicates and search conditions**	— Subclause 8.20, "<search condition>", and Subclause 8.1, "<predicate>"
35	E061-01	Comparison predicate	— Subclause 8.2, "<comparison predicate>": For supported data types, without support for <table subquery>
36	E061-02	BETWEEN predicate	— Subclause 8.3, "<between predicate>"
37	E061-03	IN predicate with list of values	— Subclause 8.4, "<in predicate>": Without support for <table subquery>
38	E061-04	LIKE predicate	— Subclause 8.5, "<like predicate>": Without `[ ESCAPE <escape character> ]`
39	E061-05	LIKE predicate: ESCAPE clause	— Subclause 8.5, "<like predicate>": With `[ ESCAPE <escape character> ]`
40	E061-06	NULL predicate	— Subclause 8.8, "<null predicate>": Without Feature F481, "Expanded NULL predicate"
41	E061-07	Quantified comparison predicate	— Subclause 8.9, "<quantified comparison predicate>": Without support for <table subquery>
42	E061-08	EXISTS predicate	— Subclause 8.10, "<exists predicate>"
43	E061-09	Subqueries in comparison predicate	— Subclause 8.2, "<comparison predicate>": For supported data types, with support for <table subquery>
44	E061-11	Subqueries in IN predicate	— Subclause 8.4, "<in predicate>": With support for <table subquery>
45	E061-12	Subqueries in quantified comparison predicate	— Subclause 8.9, "<quantified comparison predicate>": With support for <table subquery>

	Feature ID	Feature Name	Feature Description
46	E061-13	Correlated subqueries	— Subclause 8.1, "<predicate>": When a <correlation name> can be used in a <table subquery> as a correlated reference to a column in the outer query
47	E061-14	Search condition	— Subclause 8.20, "<search condition>"
48	**E071**	**Basic query expressions**	— Subclause 7.13, "<query expression>"
49	E071-01	UNION DISTINCT table operator	— Subclause 7.13, "<query expression>": With support for `UNION [ DISTINCT ]`
50	E071-02	UNION ALL table operator	— Subclause 7.13, "<query expression>": With support for `UNION ALL`
51	E071-03	EXCEPT DISTINCT table operator	— Subclause 7.13, "<query expression>": With support for `EXCEPT [ DISTINCT ]`
52	E071-05	Columns combined via table operators need not have exactly the same data type.	— Subclause 7.13, "<query expression>": Columns combined via UNION and EXCEPT need not have exactly the same data type
53	E071-06	Table operators in subqueries	— Subclause 7.13, "<query expression>": <table subquery>s can specify UNION and EXCEPT
54	**E081**	**Basic Privileges**	— Subclause 12.3, "<privileges>"
55	E081-01	SELECT privilege at the table level	— Subclause 12.3, "<privileges>": With <action> of SELECT without <privilege column list>
56	E081-02	DELETE privilege	— Subclause 12.3, "<privileges>": With <action> of DELETE
57	E081-03	INSERT privilege at the table level	— Subclause 12.3, "<privileges>": With <action> of INSERT without <privilege column list>
58	E081-04	UPDATE privilege at the table level	— Subclause 12.3, "<privileges>": With <action> of UPDATE without <privilege column list>
59	E081-05	UPDATE privilege at the column level	— Subclause 12.3, "<privileges>": With <action> of `UPDATE <left paren> <privilege column list> <right paren>`
60	E081-06	REFERENCES privilege at the table level	— Subclause 12.3, "<privileges>": with <action> of REFERENCES without <privilege column list>
61	E081-07	REFERENCES privilege at the column level	— Subclause 12.3, "<privileges>": With <action> of `REFERENCES <left paren> <privilege column list> <right paren>`

	Feature ID	Feature Name	Feature Description
62	E081-08	WITH GRANT OPTION	— Subclause 12.2, “<grant privilege statement>”: `WITH GRANT OPTION`
63	E081-09	USAGE privilege	— Subclause 12.3, “<privileges>”: With <action> of USAGE
64	E081-10	EXECUTE privilege	— Subclause 12.3, “<privileges>”: With <action> of EXECUTE
65	**E091**	**Set functions**	— Subclause 6.9, “<set function specification>”
66	E091-01	AVG	— Subclause 6.9, “<set function specification>”: With <computational operation> of AVG
67	E091-02	COUNT	— Subclause 6.9, “<set function specification>”: With <computational operation> of COUNT
68	E091-03	MAX	— Subclause 6.9, “<set function specification>”: With <computational operation> of MAX
69	E091-04	MIN	— Subclause 6.9, “<set function specification>”: With <computational operation> of MIN
70	E091-05	SUM	— Subclause 6.9, “<set function specification>”: With <computational operation> of SUM
71	E091-06	ALL quantifier	— Subclause 6.9, “<set function specification>”: With <set quantifier> of ALL
72	E091-07	DISTINCT quantifier	— Subclause 6.9, “<set function specification>”: With <set quantifier> of DISTINCT
73	**E101**	**Basic data manipulation**	— Clause 14, “Data manipulation”: <insert statement>, <delete statement: searched>, and <update statement: searched>
74	E101-01	INSERT statement	— Subclause 14.10, “<insert statement>”: When a <contextually typed table value constructor> can consist of no more than a single <contextually typed row value expression>
75	E101-03	Searched UPDATE statement	— Subclause 14.13, “<update statement: searched>”: But without support either of Feature E153, “Updatable tables with subqueries”, or Feature F221, “Explicit defaults”
76	E101-04	Searched DELETE statement	— Subclause 14.9, “<delete statement: searched>”

	Feature ID	Feature Name	Feature Description
77	**E111**	**Single row SELECT statement**	— Subclause 14.7, "<select statement: single row>": Without support of Feature F131, "Grouped operations"
78	**E121**	**Basic cursor support**	— Clause 14, "Data manipulation": <declare cursor>, <open statement>, <fetch statement>, <close statement>, <delete statement: positioned>, and <update statement: positioned>
79	E121-01	DECLARE CURSOR	— Subclause 14.1, "<declare cursor>": When each <value expression> in the <sort key> shall be a <column reference> and that <column reference> shall also be in the <select list>, and <cursor holdability> is not specified
80	E121-02	ORDER BY columns need not be in select list	— Subclause 14.1, "<declare cursor>": Extend subfeature E121-01 so that <column reference> need not also be in the <select list>
81	E121-03	Value expressions in ORDER BY clause	— Subclause 14.1, "<declare cursor>": Extend subfeature E121-01 so that the <value expression> in the <sort key> need not be a <column reference>
82	E121-04	OPEN statement	— Subclause 14.4, "<open statement>"
83	E121-06	Positioned UPDATE statement	— Subclause 14.12, "<update statement: positioned>": Without support of either Feature E153, "Updateable tables with subqueries" or Feature F221, "Explicit defaults"
84	E121-07	Positioned DELETE statement	— Subclause 14.8, "<delete statement: positioned>"
85	E121-08	CLOSE statement	— Subclause 14.6, "<close statement>"
86	E121-10	FETCH statement: implicit NEXT	— Subclause 14.5, "<fetch statement>"
87	E121-17	WITH HOLD cursors	— Subclause 14.1, "<declare cursor>": Where the <value expression> in the <sort key> need not be a <column reference> and need not be in the <select list>, and <cursor holdability> may be specified
88	**E131**	**Null value support (nulls in lieu of values)**	— Subclause 4.13, "Columns, fields, and attributes": Nullability characteristic — Subclause 6.5, "<contextually typed value specification>": <null specification>

	Feature ID	Feature Name	Feature Description
89	**E141**	**Basic integrity constraints**	— Subclause 11.6, "<table constraint definition>": As specified by the subfeatures of this feature in this table
90	E141-01	NOT NULL constraints	— Subclause 11.4, "<column definition>": With <column constraint> of NOT NULL
91	E141-02	UNIQUE constraints of NOT NULL columns	— Subclause 11.4, "<column definition>": With <unique specification> of UNIQUE for columns specified as NOT NULL — Subclause 11.7, "<unique constraint definition>": With <unique specification> of UNIQUE
92	E141-03	PRIMARY KEY constraints	— Subclause 11.4, "<column definition>": With <unique specification> of PRIMARY KEY for columns specified as NOT NULL — Subclause 11.7, "<unique constraint definition>": With <unique specification> of PRIMARY KEY
93	E141-04	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action.	— Subclause 11.4, "<column definition>": With <column constraint> of <references specification> — Subclause 11.8, "<referential constraint definition>": Where the columns in the <column name list>, if specified, shall be in the same order as the names in the <unique column list> of the applicable <unique constraint definition> and the <data type>s of the matching columns shall be the same
94	E141-06	CHECK constraints	— Subclause 11.4, "<column definition>": With <column constraint> of <check constraint definition> — Subclause 11.9, "<check constraint definition>"
95	E141-07	Column defaults	— Subclause 11.4, "<column definition>": With <default clause>
96	E141-08	NOT NULL inferred on PRIMARY KEY	— Subclause 11.4, "<column definition>", and Subclause 11.7, "<unique constraint definition>": Remove the restriction in subfeatures E141-02 and E141-03 that NOT NULL be specified along with every PRIMARY KEY and UNIQUE constraint — Subclause 11.4, "<column definition>": NOT NULL is implicit on PRIMARY KEY constraints

	Feature ID	Feature Name	Feature Description
97	E141-10	Names in a foreign key can be specified in any order	— Subclause 11.4, “<column definition>”, and Subclause 11.8, “<referential constraint definition>”: Extend subfeature E141-04 so that the columns in the <column name list>, if specified, need not be in the same order as the names in the <unique column list> of the applicable <unique constraint definition>
98	**E151**	**Transaction support**	— Clause 17, “Transaction management”: <commit statement> and <rollback statement>
99	E151-01	COMMIT statement	— Subclause 17.7, “<commit statement>”
100	E151-02	ROLLBACK statement	— Subclause 17.8, “<rollback statement>”
101	**E152**	**Basic SET TRANSACTION statement**	— Subclause 17.2, “<set transaction statement>”
102	E152-01	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	— Subclause 17.2, “<set transaction statement>”: With <transaction mode> of ISOLATION LEVEL SERIALIZABLE clause
103	E152-02	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	— Subclause 17.2, “<set transaction statement>”: with <transaction access mode> of READ ONLY or READ WRITE
104	**E153**	**Updatable queries with subqueries**	— Subclause 7.13, “<query expression>”: A <query expression> is updatable even though its <where clause> contains a <subquery>
105	**E161**	**SQL comments using leading double minus**	— Subclause 5.2, “<token> and <separator>”: <simple comment>
106	**E171**	**SQLSTATE support**	— Subclause 24.1, “SQLSTATE”
107	**E182**	**Module language**	— Clause 13, “SQL-client modules” NOTE 501 — An SQL-implementation is required to supply at least one binding to a standard host language using either module language, embedded SQL, or both.
108	**F031**	**Basic schema manipulation**	— Clause 11, “Schema definition and manipulation”: Selected facilities as indicated by the subfeatures of this Feature
109	F031-01	CREATE TABLE statement to create persistent base tables	— Subclause 11.3, “<table definition>”: Not in the context of a <schema definition>

	Feature ID	Feature Name	Feature Description
110	F031-02	CREATE VIEW statement	— Subclause 11.22, “<view definition>”: Not in the context of a <schema definition>, and without support of Feature F081, “UNION and EXCEPT in views”
111	F031-03	GRANT statement	— Subclause 12.1, “<grant statement>”: Not in the context of a <schema definition>
112	F031-04	ALTER TABLE statement: ADD COLUMN clause	— Subclause 11.10, “<alter table statement>”: The <add column definition> clause — Subclause 11.11, “<add column definition>”
113	F031-13	DROP TABLE statement: RESTRICT clause	— Subclause 11.21, “<drop table statement>”: With a <drop behavior> of RESTRICT
114	F031-16	DROP VIEW statement: RESTRICT clause	— Subclause 11.23, “<drop view statement>”: With a <drop behavior> of RESTRICT
115	F031-19	REVOKE statement: RESTRICT clause	— Subclause 12.7, “<revoke statement>”: With a <drop behavior> of RESTRICT, only where the use of this statement can be restricted to the owner of the table being dropped
116	**F041**	**Basic joined table**	— Subclause 7.7, “<joined table>”
117	F041-01	Inner join (but not necessarily the INNER keyword)	— Subclause 7.6, “<table reference>”: The <joined table> clause, but without support for subfeatures F041-02 through F041-08
118	F041-02	INNER keyword	— Subclause 7.7, “<joined table>”: <join type> of INNER
119	F041-03	LEFT OUTER JOIN	— Subclause 7.7, “<joined table>”: <outer join type> of LEFT
120	F041-04	RIGHT OUTER JOIN	— Subclause 7.7, “<joined table>”: <outer join type> of RIGHT
121	F041-05	Outer joins can be nested	— Subclause 7.7, “<joined table>”: Subfeature F041-1 extended so that a <table reference> within the <joined table> can itself be a <joined table>
122	F041-07	The inner table in a left or right outer join can also be used in an inner join	— Subclause 7.7, “<joined table>”: Subfeature F041-1 extended so that a <table name> within a nested <joined table> can be the same as a <table name> in an outer <joined table>

	Feature ID	Feature Name	Feature Description
123	F041-08	All comparison operators are supported (rather than just =)	— Subclause 7.7, "<joined table>": Subfeature F041-1 extended so that the <join condition> is not limited to a <comparison predicate> with a <comp op> of <equals operator>
124	**F051**	**Basic date and time**	— Subclause 6.1, "<data type>": <datetime type> including datetime literals, datetime comparisons, and datetime conversions
125	F051-01	DATE data type (including support of DATE literal)	— Subclause 5.3, "<literal>": The <date literal> form of <datetime literal> — Subclause 6.1, "<data type>": The DATE <datetime type> — Subclause 6.30, "<datetime value expression>": For values of type DATE
126	F051-02	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0.	— Subclause 5.3, "<literal>": The <time literal> form of <datetime literal>, where the value of <unquoted time string> is simply <time value> that does not include the optional <time zone interval> — Subclause 6.1, "<data type>": The TIME <datetime type> without the <with or without time zone> clause — Subclause 6.30, "<datetime value expression>": For values of type TIME
127	F051-03	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6.	— Subclause 5.3, "<literal>": The <timestamp literal> form of <datetime literal>, where the value of <unquoted timestamp string> is simply <time value> that does not include the optional <time zone interval> — Subclause 6.1, "<data type>": The TIMESTAMP <datetime type> without the <with or without time zone> clause — Subclause 6.30, "<datetime value expression>": For values of type TIMESTAMP
128	F051-04	Comparison predicate on DATE, TIME, and TIMESTAMP data types	— Subclause 8.2, "<comparison predicate>": For comparison between values of the following types: DATE and DATE, TIME and TIME, TIMESTAMP and TIMESTAMP, DATE and TIMESTAMP, and TIME and TIMESTAMP

	Feature ID	Feature Name	Feature Description
129	F051-05	Explicit CAST between date-time types and character string types	— Subclause 6.12, "<cast specification>": If support for Feature F201, "CAST function" is available, then CASTing between the following types: from character string to DATE, TIME, and TIMESTAMP; from DATE to DATE, TIMESTAMP, and character string; from TIME to TIME, TIMESTAMP, and character string; from TIMESTAMP to DATE, TIME, TIMESTAMP, and character string
130	F051-06	CURRENT_DATE	— Subclause 6.31, "<datetime value function>": The <current date value function> — Subclause 6.30, "<datetime value expression>": When the value is a <current date value function>
131	F051-07	LOCALTIME	— Subclause 6.31, "<datetime value function>": The <current local time value function> — Subclause 6.30, "<datetime value expression>": When the value is a <current local time value function> — Subclause 11.5, "<default clause>": LOCALTIME option of <datetime value function>
132	F051-08	LOCALTIMESTAMP	— Subclause 6.31, "<datetime value function>": The <current local timestamp value function> — Subclause 6.30, "<datetime value expression>": When the value is a <current local timestamp value function> — Subclause 11.5, "<default clause>": LOCALTIMESTAMP option of <datetime value function>
133	**F081**	**UNION and EXCEPT in views**	— Subclause 11.22, "<view definition>": A <query expression> in a <view definition> may specify UNION, UNION ALL, and/or EXCEPT
134	**F131**	**Grouped operations**	— A *grouped view* is a view whose <query expression> contains a <group by clause>
135	F131-01	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	— Subclause 7.4, "<table expression>": Even though a table in the <from clause> is a grouped view, the <where clause>, <group by clause>, and <having clause> may be specified
136	F131-02	Multiple tables supported in queries with grouped views	— Subclause 7.5, "<from clause>": Even though a table in the <from clause> is a grouped view, the <from clause> may specify more than one <table reference>

	Feature ID	Feature Name	Feature Description
137	F131-03	Set functions supported in queries with grouped views	— Subclause 7.12, "<query specification>": Even though a table in the <from clause> is a grouped view, the <select list> may specify a <set function specification>
138	F131-04	Subqueries with GROUP BY and HAVING clauses and grouped views	— Subclause 7.15, "<subquery>": A <subquery> in a <comparison predicate> is allowed to contain a <group by clause> and/or a <having clause> and/or it may identify a grouped view
139	F131-05	Single row SELECT with GROUP BY and HAVING clauses and grouped views	— Subclause 14.7, "<select statement: single row>": The table in a <from clause> can be a grouped view — Subclause 14.7, "<select statement: single row>": The <table expression> may specify a <group by clause> and/or a <having clause>
140	**F181**	**Multiple module support** NOTE 502 — The ability to associate multiple host compilation units with a single SQL-session at one time.	— Subclause 13.1, "<SQL-client module definition>":An SQL-agent can be associated with more than one <SQL-client module definition> With this feature, it is possible to compile <SQL-client module definition>s or <embedded SQL host program>s separately and rely on the SQL-implementation to "link" the together properly at execution time. To ensure portability, applications should adhere to the following limitations: —Avoid linking modules having cursors with the same <cursor name>. —Avoid linking modules that prepare statements using the same <SQL statement name>. —Avoid linking modules that allocate descriptors with the same <descriptor name>. —Assume that the scope of an <embedded exception declaration> is a single compilation unit. —Assume that an <embedded variable name> can be referenced only in the same compilation unit in which it is declared.
141	**F201**	**CAST function** NOTE 503 — This means the support of CAST, where relevant, among all supported data types.	— Subclause 6.12, "<cast specification>": For all supported data types — Subclause 6.25, "<value expression>": <cast specification>
142	**F221**	**Explicit defaults**	— Subclause 6.5, "<contextually typed value specification>": <default specification> NOTE 504 — Including its use in UPDATE and INSERT statements.

	Feature ID	Feature Name	Feature Description
143	**F261**	**CASE expression**	— Subclause 6.25, “<value expression>”: <case expression>
144	F261-01	Simple CASE	— Subclause 6.11, “<case expression>”: The <simple case> variation
145	F261-02	Searched CASE	— Subclause 6.11, “<case expression>”: The <searched case> variation
146	F261-03	NULLIF	— Subclause 6.11, “<case expression>”: The NULLIF <case abbreviation>
147	F261-04	COALESCE	— Subclause 6.11, “<case expression>”: The COALESCE <case abbreviation>
148	**F311**	**Schema definition statement**	— Subclause 11.1, “<schema definition>”
149	F311-01	CREATE SCHEMA	— Subclause 11.1, “<schema definition>”: Support for circular references in that <referential constraint definition>s in two different <table definition>s may reference columns in the other table
150	F311-02	CREATE TABLE for persistent base tables	— Subclause 11.1, “<schema definition>”: A <schema element> that is a <table definition> — Subclause 11.3, “<table definition>”: In the context of a <schema definition>
151	F311-03	CREATE VIEW	— Subclause 11.1, “<schema definition>”: A <schema element> that is a <view definition> — Subclause 11.22, “<view definition>”: In the context of a <schema definition> without the WITH CHECK OPTION clause and without support of Feature F081, “UNION and EXCEPT in views”
152	F311-04	CREATE VIEW: WITH CHECK OPTION	— Subclause 11.22, “<view definition>”: The WITH CHECK OPTION clause, in the context of a <schema definition>, but without support of Feature F081, “UNION and EXCEPT in views”
153	F311-05	GRANT statement	— Subclause 11.1, “<schema definition>”: A <schema element> that is a <grant statement> — Subclause 12.1, “<grant statement>”: In the context of a <schema definition>
154	**F471**	**Scalar subquery values**	— Subclause 6.25, “<value expression>”: A <value expression primary> can be a <scalar subquery>

	Feature ID	Feature Name	Feature Description
155	**F481**	**Expanded NULL predicate**	— Subclause 8.8, “<null predicate>”: The <row value expression> can be something other than a <column reference>
156	**F812**	**Basic flagging**	— Part 1, Subclause 8.5, “SQL flagger”: With “level of flagging” specified to be Core SQL Flagging and “extent of checking” specified to be Syntax Only NOTE 505 — This form of flagging identifies vendor extensions and other non-standard SQL by checking syntax only without requiring access to the catalog information.
157	**S011**	**Distinct data types**	— Subclause 11.41, “<user-defined type definition>”: When <representation> is <predefined type> — Subclause 11.49, “<drop data type statement>”
158	**T321**	**Basic SQL-invoked routines**	— Subclause 11.50, “<SQL-invoked routine>” — Subclause 11.52, “<drop routine statement>” — If Feature T041, “Basic LOB data type support”, is supported, then the <locator indication> clause shall also be supported NOTE 506 — “Routine” is the collective term for functions, methods, and procedures. This feature requires a conforming SQL-implementation to support both user-defined functions and user-defined procedures. An SQL-implementation that conforms to Core SQL shall support at least one language for writing routines; that language may be SQL. If the language is SQL, then the basic specification capability in Core SQL is the ability to specify a one-statement routine. Support for overloaded functions and procedures is not part of Core SQL.
159	T321-01	User-defined functions with no overloading	— Subclause 11.50, “<SQL-invoked routine>”: With <function specification>
160	T321-02	User-defined stored procedures with no overloading	— Subclause 11.50, “<SQL-invoked routine>”: With <SQL-invoked procedure>
161	T321-03	Function invocation	— Subclause 6.4, “<value specification> and <target specification>”: With a <value expression primary> that is a <routine invocation> — Subclause 10.4, “<routine invocation>”: For user-defined functions
162	T321-04	CALL statement	— Subclause 10.4, “<routine invocation>”: Used by <call statement>s — Subclause 16.1, “<call statement>”

	Feature ID	Feature Name	Feature Description
163	T321-05	RETURN statement	— Subclause 16.2, "<return statement>", if the SQL-implementation supports SQL routines
164	**T631**	**IN predicate with one list element**	— Subclause 8.4, "<in predicate>": <in value list> containing exactly one <row value expression>
[1] A conforming SQL-implementation is required (by Clause 8, "Conformance", in ISO/IEC 9075-1) to support at least one embedded language or to support the SQL-client module binding for at least one host language.			

Table 37, "Feature taxonomy for optional features", does not provide definitions of the features; the definition of those features is found in the Conformance Rules that are further summarized in Annex A, "SQL Conformance Summary".

Table 37 — Feature taxonomy for optional features

	Feature ID	Feature Name
1	**B011**	**Embedded Ada**
2	**B012**	**Embedded C**
3	**B013**	**Embedded COBOL**
4	**B014**	**Embedded Fortran**
5	**B015**	**Embedded MUMPS**
6	**B016**	**Embedded Pascal**
7	**B017**	**Embedded PL/I**
8	**B021**	**Direct SQL**
9	**B031**	**Basic dynamic SQL**
10	**B032**	**Extended dynamic SQL**
11	B032-01	<describe input statement>
12	**B033**	**Untyped SQL-invoked function arguments**
13	**B034**	**Dynamic specification of cursor attributes**
14	**B041**	**Extensions to embedded SQL exception declarations**
15	**B051**	**Enhanced execution rights**
16	**B111**	**Module language Ada**

	Feature ID	Feature Name
17	**B112**	**Module language C**
18	**B113**	**Module language COBOL**
19	**B114**	**Module language Fortran**
20	**B115**	**Module language MUMPS**
21	**B116**	**Module language Pascal**
22	**B117**	**Module language PL/I**
23	**B121**	**Routine language Ada**
24	**B122**	**Routine language C**
25	**B123**	**Routine language COBOL**
26	**B124**	**Routine language Fortran**
27	**B125**	**Routine language MUMPS**
28	**B126**	**Routine language Pascal**
29	**B127**	**Routine language PL/I**
30	**B128**	**Routine language SQL**
31	**F032**	**CASCADE drop behavior**
32	**F033**	**ALTER TABLE statement: DROP COLUMN clause**
33	**F034**	**Extended REVOKE statement**
34	F034-01	REVOKE statement performed by other than the owner of a schema object
35	F034-02	REVOKE statement: GRANT OPTION FOR clause
36	F034-03	REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION
37	**F052**	**Intervals and datetime arithmetic**
38	**F053**	**OVERLAPS predicate**
39	**F111**	**Isolation levels other than SERIALIZABLE**
40	F111-01	READ UNCOMMITTED isolation level
41	F111-02	READ COMMITTED isolation level

	Feature ID	Feature Name
42	F111-03	REPEATABLE READ isolation level
43	**F121**	**Basic diagnostics management**
44	F121-01	GET DIAGNOSTICS statement
45	F121-02	SET TRANSACTION statement: DIAGNOSTICS SIZE clause
46	**F122**	**Enhanced diagnostics management**
47	**F123**	**All diagnostics**
48	**F171**	**Multiple schemas per user**
49	**F191**	**Referential delete actions**
50	**F222**	**INSERT statement: DEFAULT VALUES clause**
51	**F231**	**Privilege tables**
52	F231-01	TABLE_PRIVILEGES view
53	F231-02	COLUMN_PRIVILEGES view
54	F231-03	USAGE_PRIVILEGES view
55	**F251**	**Domain support**
56	**F262**	**Extended CASE expression**
57	**F263**	**Comma-separated predicates in simple CASE expression**
58	**F271**	**Compound character literals**
59	**F281**	**LIKE enhancements**
60	**F291**	**UNIQUE predicate**
61	**F301**	**CORRESPONDING in query expressions**
62	**F302**	**INTERSECT table operator**
63	F302-01	INTERSECT DISTINCT table operator
64	F302-02	INTERSECT ALL table operator
65	**F304**	**EXCEPT ALL table operator**
66	**F312**	**MERGE statement**

	Feature ID	Feature Name
67	**F321**	**User authorization**
68	**F341**	**Usage tables**
69	**F361**	**Subprogram support**
70	**F381**	**Extended schema manipulation**
71	F381-01	ALTER TABLE statement: ALTER COLUMN clause
72	F381-02	ALTER TABLE statement: ADD CONSTRAINT clause
73	F381-03	ALTER TABLE statement: DROP CONSTRAINT clause
74	**F391**	**Long identifiers**
75	**F392**	**Unicode escapes in identifiers**
76	**F393**	**Unicode escapes in literals**
77	**F394**	**Optional normal form specification**
78	**F401**	**Extended joined table**
79	F401-01	NATURAL JOIN
80	F401-02	FULL OUTER JOIN
81	F401-04	CROSS JOIN
82	**F402**	**Named column joins for LOBs, arrays, and multisets**
83	**F403**	**Partitioned join tables**
84	**F411**	**Time zone specification**
85	**F421**	**National character**
86	**F431**	**Read-only scrollable cursors**
87	F431-01	FETCH with explicit NEXT
88	F431-02	FETCH FIRST
89	F431-03	FETCH LAST
90	F431-04	FETCH PRIOR
91	F431-05	FETCH ABSOLUTE

	Feature ID	Feature Name
92	F431-06	FETCH RELATIVE
93	**F441**	**Extended set function support**
94	**F442**	**Mixed column references in set functions**
95	**F451**	**Character set definition**
96	**F461**	**Named character sets**
97	**F491**	**Constraint management**
98	**F502**	**Enhanced documentation tables**
99	F502-01	SQL_SIZING_PROFILES view
100	F502-02	SQL_IMPLEMENTATION_INFO view
101	F502-03	SQL_PACKAGES view
102	**F521**	**Assertions**
103	**F531**	**Temporary tables**
104	**F555**	**Enhanced seconds precision**
105	**F561**	**Full value expressions**
106	**F571**	**Truth value tests**
107	**F591**	**Derived tables**
108	**F611**	**Indicator data types**
109	**F641**	**Row and table constructors**
110	**F651**	**Catalog name qualifiers**
111	**F661**	**Simple tables**
112	**F671**	**Subqueries in CHECK**
113	**F672**	**Retrospective check constraints**
114	**F690**	**Collation support**
115	**F692**	**Enhanced collation support**
116	**F693**	**SQL-session and client module collations**

	Feature ID	Feature Name
117	**F695**	**Translation support**
118	**F701**	**Referential update actions**
119	**F711**	**ALTER domain**
120	**F721**	**Deferrable constraints**
121	**F731**	**INSERT column privileges**
122	**F741**	**Referential MATCH types**
123	**F751**	**View CHECK enhancements**
124	**F761**	**Session management**
125	**F762**	**CURRENT_CATALOG**
126	**F763**	**CURRENT_SCHEMA**
127	**F771**	**Connection management**
128	**F781**	**Self-referencing operations**
129	**F791**	**Insensitive cursors**
130	**F801**	**Full set function**
131	**F813**	**Extended flagging** — Part 1, Subclause 8.5, "SQL flagger": With "level of flagging" specified to be Core SQL Flagging and "extent of checking" specified to be Catalog Lookup
132	**F821**	**Local table references**
133	**F831**	**Full cursor update**
134	F831-01	Updateable scrollable cursors
135	F831-02	Updateable ordered cursors
136	**F841**	**LIKE_REGEX predicate**
137	**F842**	**OCCURRENCES_REGEX function**
138	**F843**	**POSITION_REGEX function**
139	**F844**	**SUBSTRING_REGEX function**
140	**F845**	**TRANSLATE_REGEX function**

	Feature ID	Feature Name
141	**F846**	**Octet support in regular expression operators**
142	**F847**	**Nonconstant regular expressions**
143	**S023**	**Basic structured types**
144	**S024**	**Enhanced structured types**
145	**S025**	**Final structured types**
146	**S026**	**Self-referencing structured types**
147	**S027**	**Create method by specific method name**
148	**S028**	**Permutable UDT options list**
149	**S041**	**Basic reference types**
150	**S043**	**Enhanced reference types**
151	**S051**	**Create table of type**
152	**S071**	**SQL paths in function and type name resolution**
153	**S081**	**Subtables**
154	**S091**	**Basic array support**
155	S091-01	Arrays of built-in data types
156	S091-02	Arrays of distinct types
157	S091-03	Array expressions
158	**S092**	**Arrays of user-defined types**
159	**S094**	**Arrays of reference types**
160	**S095**	**Array constructors by query**
161	**S096**	**Optional array bounds**
162	**S097**	**Array element assignment**
163	**S111**	**ONLY in query expressions**
164	**S151**	**Type predicate**
165	**S161**	**Subtype treatment**

	Feature ID	Feature Name
166	**S162**	**Subtype treatment for references**
167	**S201**	**SQL-invoked routines on arrays**
168	S201-01	Array parameters
169	S201-02	Array as result type of functions
170	**S202**	**SQL-invoked routines on multisets**
171	**S211**	**User-defined cast functions**
172	**S231**	**Structured type locators**
173	**S232**	**Array locators**
174	**S233**	**Multiset locators**
175	**S241**	**Transform functions**
176	**S242**	**Alter transform statement**
177	**S251**	**User-defined orderings**
178	**S261**	**Specific type method**
179	**S271**	**Basic multiset support**
180	**S272**	**Multisets of user-defined types**
181	**S274**	**Multisets of reference types**
182	**S275**	**Advanced multiset support**
183	**S281**	**Nested collection types**
184	**S291**	**Unique constraint on entire row**
185	**T011**	**Timestamp in Information Schema**
186	**T021**	**BINARY and VARBINARY data types**
187	**T022**	**Advanced BINARY and VARBINARY data type support**
188	**T024**	**Compound binary literals**
189	**T024**	**Spaces in binary literals**
190	**T031**	**BOOLEAN data type**

	Feature ID	Feature Name
191	**T041**	**Basic LOB data type support**
192	T041-01	BLOB data type — Subclause 5.2, “<token> and <separator>”: The <reserved word>s BINARY, BLOB, LARGE, and OBJECT — Subclause 5.3, “<literal>”: <binary string literal> — Subclause 6.1, “<data type>”: The BINARY LARGE OBJECT data type — Subclause 6.28, “<string value expression>”: For values of type BINARY LARGE OBJECT — Subclause 13.6, “Data type correspondences”: Type correspondences for BINARY LARGE OBJECT for all supported languages
193	T041-02	CLOB data type — Subclause 5.2, “<token> and <separator>”: The <reserved word>s CHARACTER, CLOB, LARGE, and OBJECT — Subclause 6.1, “<data type>”: The CHARACTER LARGE OBJECT data type — Subclause 6.28, “<string value expression>”: For values of type CHARACTER LARGE OBJECT — Subclause 13.6, “Data type correspondences”: Type correspondences for CHARACTER LARGE OBJECT for all supported languages — The implicit casting among the fixed-length and variable-length character string types supported by subfeature E021-10 is extended to support the character large object type
194	T041-03	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types — Subclause 6.27, “<numeric value function>”: The <position expression> for expressions of type BINARY LARGE OBJECT and CHARACTER LARGE OBJECT — Subclause 6.27, “<numeric value function>”: The <char length expression> for expressions of type CHARACTER LARGE OBJECT — Subclause 6.27, “<numeric value function>”: The <octet length expression> for expressions of type BINARY LARGE OBJECT and CHARACTER LARGE OBJECT — Subclause 6.29, “<string value function>”: The <fold> function for expressions of type CHARACTER LARGE OBJECT — Subclause 6.29, “<string value function>”: The <trim function> for expressions of type CHARACTER LARGE OBJECT — Subclause 6.29, “<string value function>”: The <binary trim function> for expressions of type BINARY LARGE OBJECT — Subclause 6.29, “<string value function>”: The <character substring function> for expressions of type CHARACTER LARGE OBJECT — Subclause 6.29, “<string value function>”: The <binary substring function> for expressions of type BINARY LARGE OBJECT

	Feature ID	Feature Name
195	T041-04	Concatenation of LOB data types — Subclause 6.28, “<string value expression>”: The <concatenation> expression for expressions of type CHARACTER LARGE OBJECT — Subclause 6.28, “<string value expression>”: The <binary concatenation> expression for expressions of type BINARY LARGE OBJECT
196	T041-05	LOB locator: non-holdable — Subclause 13.3, “<externally-invoked procedure>”: <locator indication> — Subclause 14.16, “<free locator statement>”
197	**T042**	**Extended LOB data type support**
198	**T043**	**Multiplier T**
199	**T044**	**Multiplier P**
200	**T051**	**Row types**
201	**T052**	**MAX and MIN for row types**
202	**T053**	**Explicit aliases for all-fields reference**
203	**T061**	**UCS support**
204	**T071**	**BIGINT data type**
205	**T111**	**Updatable joins, unions, and columns**
206	**T121**	**WITH (excluding RECURSIVE) in query expression**
207	**T122**	**WITH (excluding RECURSIVE) in subquery**
208	**T131**	**Recursive query**
209	**T132**	**Recursive query in subquery**
210	**T141**	**SIMILAR predicate**
211	**T151**	**DISTINCT predicate**
212	**T152**	**DISTINCT predicate with negation**
213	**T171**	**LIKE clause in table definition**
214	**T172**	**AS subquery clause in table definition**
215	**T173**	**Extended LIKE clause in table definition**
216	**T174**	**Identity columns**

	Feature ID	Feature Name
217	**T175**	**Generated columns**
218	**T176**	**Sequence generator support**
219	**T191**	**Referential action RESTRICT**
220	**T201**	**Comparable data types for referential constraints**
221	**T211**	**Basic trigger capability**
222	T211-01	Triggers activated on UPDATE, INSERT, or DELETE of one base table.
223	T211-02	BEFORE triggers
224	T211-03	AFTER triggers
225	T211-04	FOR EACH ROW triggers
226	T211-05	Ability to specify a search condition that shall be *True* before the trigger is invoked.
227	T211-06	Support for run-time rules for the interaction of triggers and constraints.
228	T211-07	TRIGGER privilege
229	T211-08	Multiple triggers for the same event are executed in the order in which they were created in the catalog.
230	**T212**	**Enhanced trigger capability**
231	**T231**	**Sensitive cursors**
232	**T241**	**START TRANSACTION statement**
233	**T251**	**SET TRANSACTION statement: LOCAL option**
234	**T261**	**Chained transactions**
235	**T271**	**Savepoints**
236	**T272**	**Enhanced savepoint management**
237	**T281**	**SELECT privilege with column granularity**
238	**T301**	**Functional dependencies**
239	**T312**	**OVERLAY function**
240	**T322**	**Overloading of SQL-invoked functions and procedures**
241	**T323**	**Explicit security for external routines**

	Feature ID	Feature Name
242	T324	Explicit security for SQL routines
243	T325	Qualified SQL parameter references
244	T326	Table functions
245	T331	Basic roles
246	T332	Extended roles
247	T351	Bracketed SQL comments (/*...*/ comments)
248	T431	Extended grouping capabilities
249	T432	Nested and concatenated GROUPING SETS
250	T433	Multiargument GROUPING function
251	T434	GROUP BY DISINCT
252	T441	ABS and MOD functions
253	T461	Symmetric BETWEEN predicate
254	T471	Result sets return value
255	T491	LATERAL derived table
256	T501	Enhanced EXISTS predicate
257	T511	Transaction counts
258	T551	Optional key words for default syntax
259	T561	Holdable locators
260	T571	Array-returning external SQL-invoked functions
261	T572	Multiset-returning external SQL-invoked functions
262	T581	Regular expression substring function
263	T591	UNIQUE constraints of possibly null columns
264	T601	Local cursor references
265	T611	Elementary OLAP operations
266	T612	Advanced OLAP operations

	Feature ID	Feature Name
267	T613	Sampling
268	T621	Enhanced numeric functions
269	T641	Multiple column assignment
270	T651	SQL-schema statements in SQL routines
271	T652	SQL-dynamic statements in SQL routines
272	T653	SQL-schema statements in external routines
273	T654	SQL-dynamic statements in external routines
274	T655	Cyclically dependent routines

(Blank page)

Annex G
(informative)

Defect Reports not addressed in this edition of ISO/IEC 9075

Each entry in this Annex describes a reported defect in the previous edition of this part of ISO/IEC 9075 that remains in this edition.

1) **Subclause 10.4, "<routine invocation>"**:

   There is no definition of how to pass values of type BOOLEAN or of large object types as arguments to invocations of external routines. More generally, the question of how to convert a value of any SQL type to a value of an appropriate host language type at the interface to an SQL-invoked routine is not addressed. The rules in Subclause 13.4, "Calls to an <externally-invoked procedure>", are appropriate, but they are not referenced by the rules of Subclause 10.4, "<routine invocation>".

2) **Subclause 21.1, "<embedded SQL host program>"**:

   SR 21)h)i)6) and SR 21)l)i)3)B)VI) both refer to the SQL data type that corresponds to a given host language data type, as determined by application of the rules in Subclause 13.6, "Data type correspondences". These two syntax rules are sometimes ambiguous, because Subclause 13.6, "Data type correspondences" does not always give exactly one SQL data type for a given host language type, as can be seen by inspection of the data type correspondence tables given in that Subclause. For example, Table 17, "Data type correspondences for C", in which the C data type "pointer to long" maps to both INTEGER and BOOLEAN.

3) **Subclause 17.4, "<set constraints mode statement>"**:

   There are several problems with the deferred constraint checking specified by use of the keyword DEFERRED:

   a) Exactly when <referential action>s of deferred referential constraints are processed is not precisely specified.

   b) When referential constraint checking is immediate and execution of an SQL-statement causes rows to be deleted, those rows are merely "marked for deletion" and not actually deleted until all constraint checking has been done. This ensures the correct processing of <referential action>s, such as ON DELETE SET DEFAULT, that cause changes to SQL-data. When the checking of some referential constraint has been deferred and the mode of that constraint is set to IMMEDIATE, it can happen that a row whose presence is needed to ensure the correct processing of a <referential action> has been actually deleted, as a result of the successful prior execution of some SQL-data change statement.

      For example, consider:

```
CREATE TABLE T1 ( A INTEGER, PRIMARY KEY ( A ) ) ;
CREATE TABLE T2 ( A INTEGER,
  CONSTRAINT C1
    FOREIGN KEY ( A ) REFERENCES T2
    ON UPDATE CASCADE
    DEFERRABLE) ;
```

```
INSERT INTO T1 VALUES ( 1 ) ;
INSERT INTO T2 VALUES ( 1 ) ;
```

Now VALUES (1) is a matching row in T2 for the only row in T1.

```
SET CONSTRAINTS C1 DEFERRED ;
UPDATE T1 SET A = 9 ;
```

The processing of the UPDATE statement causes the only row in T1 to change from VALUES (1) to VALUES (9). The update to that row in T1 is supposed to be propagated to the only row in T2 under the rule for constraint C1, but processing of constraint C1 is deferred and does not take place at this time.

```
SET CONSTRAINTS C1 IMMEDIATE ;
```

Constraint C1 is processed now, but, there now being no row in T1 for which the VALUES (1) in T2 is a matching row, no change to T2 will take place under the General Rules of Subclause 11.8, "<referential constraint definition>". Thus, the constraint is violated in spite of the existence of the <referential action> that is expected to prevent this from happening. Moreover, the unmatched row in T2 remains even after the mode of C1 has been set to IMMEDIATE, for a <set constraints mode statement> does not make any changes to SQL-data that might be canceled under the General Rules of Subclause 13.5, "<SQL procedure statement>".

c) Various problems have been noted with the possible interaction of deferred processing of referential constraints and triggers. For example, consider:

```
CREATE TABLE T1 ( A INTEGER, PRIMARY KEY ( A ) ) ;
CREATE TABLE T2 (A INTEGER,
  PRIMARY KEY ( A ),
  CONSTRAINT C1
    FOREIGN KEY ( A ) REFERENCES T1
      ON DELETE CASCADE
      DEFERRABLE ) ;
CREATE TABLE T3 ( A INTEGER, PRIMARY KEY ( A ) ) ;
CREATE TRIGGER TR1 AFTER DELETE ON T2
  FOR EACH STATEMENT
    INSERT INTO T3 VALUES ( ( SELECT COUNT(*) FROM T3 ) + 1 ) ;
INSERT INTO T1 VALUES ( 1 ), ( 2 ) ;
INSERT INTO T2 VALUES ( 1 ), ( 2 ) ;
DELETE FROM T1 WHERE A = 1 ;
```

Because constraint checking is immediate, this deletion from T1 will cause the same row, VALUES (1), to be deleted from T2, thus activating trigger TR1 and causing VALUES (1) to be inserted into T3.

```
DELETE FROM T1 WHERE A = 2 ;
```

Likewise, this deletion will cause the row VALUES (2) to be deleted from T2, thus activating trigger TR1 and causing VALUES (1) to be inserted into T3..

Likewise, this deletion will cause the row VALUES (2) to be deleted from T2, thus activating trigger TR1 for a second time and causing VALUES (2) to be inserted into T3.

But if SET CONSTRAINTS C1 DEFERRED is executed immediately before those two deletions, then the deletions from T1 will not be propagated to T2 and so trigger TR1 will not be activated and T3 will not be updated. When SET CONSTRAINTS C1 IMMEDIATE is subsequently executed, even if the

problem illustrated in point b) is addressed so that the deletions are somehow now propagated to T2 after all, it is not clear how many activations of trigger TR1 will take place (nor, if there are more than one, in what order those activations will take place).

(Blank page)

Bibliography

[1] *POSIX* ISO/IEC 9945-1:2002, *Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: Base definitions*.

(Blank page)

Index

Index entries appearing in **boldface** indicate the page where the word, phrase, or BNF nonterminal was defined; index entries appearing in *italics* indicate a page where the BNF nonterminal was used in a Format; and index entries appearing in roman type indicate a page where the word, phrase, or BNF nonterminal was used in a heading, Function, Syntax Rule, Access Rule, General Rule, Leveling Rule, Table, or other descriptive text.

— A —

A • *144*, *438*

ABS • *145*, *254*, 263, *295*, 1185

ABSOLUTE • *144*, *843*, 889, 890

<absolute value expression> • 32, *253*, **254**, 257, 259, 263, 1185

ACTION • ?, ?, ?, *144*, *583*, 585, 820

<action> • 117, 118, 119, 548, 754, 756, 758, 759, **761**, 763, 769, 770, 785, 786, 1136, 1150, 1155, 1157, 1161, 1180, 1182, 1226, 1227

active SQL-transaction • 921, 923, 940, 945, 946, 1091, 1119

<actual identifier> • **159**

ADA • *144*, 476, *501*, 517, 518, 715, 722, 725, 793, 794, 796, 798, 810, 812, 813, 1034, 1133, 1134

<Ada array locator variable> • **1046**, 1049, 1050, 1165

<Ada assignment operator> • **1045**

<Ada BINARY variable> • **1046**, 1047, 1050, 1170

<Ada BLOB locator variable> • **1046**, 1048, 1050, 1173

<Ada BLOB variable> • **1046**, 1047, 1050, 1173

<Ada CLOB locator variable> • **1046**, 1048, 1050, 1173

<Ada CLOB variable> • **1046**, 1047, 1050, 1173

<Ada derived type specification> • *1045*, **1046**, 1047

<Ada host identifier> • *1031*, **1045**, 1046, 1047, 1049

<Ada initial value> • **1045**, 1049

<Ada multiset locator variable> • **1046**, 1049, 1050, 1166

<Ada qualified type specification> • **1045**, 1047, 1050, 1177

<Ada REF variable> • **1046**, 1049, 1050, 1158

<Ada type specification> • **1045**

<Ada unqualified type specification> • **1045**, 1050, 1177

<Ada user-defined type locator variable> • **1046**, 1048, 1051, 1164

<Ada user-defined type variable> • **1046**, 1048, 1050, 1167

<Ada VARBINARY variable> • **1046**, 1047, 1050, 1171

<Ada variable definition> • *1030*, **1045**, 1047, 1049

ADD • *144*, *595*, *600*, *606*, *631*, 634, *677*, *681*, *687*, *742*

<add attribute definition> • 86, *676*, **677**, 678

<add column definition> • *594*, **595**, 1231

<add column scope clause> • *597*, **600**, 1140, 1160

<add domain constraint definition> • *628*, **631**, 1150

<add original method specification> • *676*, **681**, 686

<add overriding method specification> • *676*, **687**, 691

<add table constraint definition> • 582, *594*, **606**, 1140

<add transform element list> • *741*, **742**, 743

additional result sets returned • 1120

ADMIN • 120, *144*, 763, *765*, *766*, *769*, 773, 774, 783, 1184

AFTER • 71, 131, 132, 135, *144*, *254*, 508, *653*, 915, 1252

<aggregate function> • *202*, 203, *204*, 206, 207, 227, 249, 366, 469, 473, **535**, 536, 545, 1144, 1212

aggregated column reference • 202

ALL • ?, 50, 63, 64, *145*, *271*, 274, 282, *304*, 305, 336, 341, *371*, 374, 375, 377, 379, 381, 383, 384, *421*, *535*, 536, 543, 611, 621, 697, *746*, 747, 759, *761*, 762, 785, 898, 905, 909, 912, *927*, 931, *940*, *1096*, 1139, 1207, 1214, 1215

<all> • **421**

<all fields column name list> • **361**, 363, 370, 1176

<all fields reference> • **361**, 362, 363, 369, 1176

<all info target> • **1096**, 1097, 1110

<all information> • *1095*, **1096**, 1109, 1111, 1137

<all qualifier> • **1096**, 1110

ALLOCATE • ?, ?, *145*, *967*, *1011*

<allocate cursor statement> • ?, ?, ?, ?, ?, 10, 84, 91, 97, 99, 100, 105, 109, 112, 114, 115, 168, 525, *817*, 837, 838, **1011**, 1012, 1013, 1100, 1132, 1186, 1206

<allocate descriptor statement> • 84, 107, 109, 168, *817*, **967**, 968, 972, 1100, 1130, 1206

ALTER • *145*, *594*, *597*, 602, 605, 608, 612, 624, *628*, 629, 630, 631, 632, 634, 651, *676*, *724*, 737, *741*, *751*, 784, 785, 1135, 1150

<alter column action> • **597**

<alter column definition> • 575, *594*, **597**, 598, 599, 600, 601, 603, 1140, 1179

<alter domain action> • **628**

<alter domain statement> • 103, 165, 575, 592, 625, **628**, 629, 630, 631, 632, 784, *816*, 1100, 1150

<alter group> • **741**, 742, 744

<alter identity column option> • **603**

<alter identity column specification> • *597*, **603**, 1179

<alter routine behavior> • **724**

<alter routine characteristic> • **724**

<alter routine characteristics> • **724**, 725

<alter routine statement> • 86, 104, **724**, 726, *815*, 1100, 1140, 1141

<alter sequence generator option> • **751**

<alter sequence generator options> • 494, 603, **751**

<alter sequence generator restart option> • 494, 495, *603*, **751**

<alter sequence generator statement> • 80, 104, **751**, *816*, 1100, 1180

<alter table action> • **594**

<alter table statement> • 103, 164, 571, 572, 573, 579, 581, 582, 583, 592, **594**, 595, 596, 597, 600, 601, 602, 604, 605, 606, 607, 608, 612, 624, 634, 651, 737, 784, 785, *815*, 1100, 1231

<alter transform action> • **741**

<alter transform action list> • **741**

<alter transform statement> • 104, **741**, 742, 744, *816*, 1100, 1168

<alter type action> • **676**

<alter type statement> • 86, 104, **676**, 677, 678, 679, 681, 687, 692, *816*, 1100, 1156, 1157

ALWAYS • 59, *144*, 562, *570*, 573

ambiguous cursor name • 978, 1114

<ampersand> • *137*, **138**, *142*, 147, *150*, 154, *1029*, 1031

AND • ?, ?, ?, ?, 20, 33, 71, 73, 126, *145*, 205, 267, 280, *296*, 297, 298, 299, 334, *352*, 353, *402*, 431, 544, 581, 735, *789*, 791, 793, *931*, 932, *933*, 934, *1030*, 1133

ANY • 64, *145*, 404, *421*, *535*, 537, 540, 544, 1172

<approximate numeric literal> • 30, **151**, 155, 157, 218, 220, 543, 1195, 1222

<approximate numeric type> • 30, 31, **172**, 176, 180, 995, 1196, 1197, 1222

ARE • *145*, *794*, *1030*

ARRAY • ?, ?, 13, 47, 48, 49, 97, *145*, *173*, 181, *193*, 194, 216, *302*, 323, 386, 387, 459, 479, 485, 575, 808, 824, 826, 827, 829, 830, 831, 859, 864, 877, 959, 960, 972, 974, 981, 995, 998, 1004, 1005, *1077*, 1078, 1079, 1080, 1081, 1161

<array concatenation> • 49, **300**, 301

array data, right truncation • 216, 301, 303, 447, 452, 1115, 1120

<array element> • **302**, 303, 981

array element error • 245, 477, 524, 846, 851, 879, 1115

<array element list> • **302**, 303, 981

<array element reference> • 49, *185*, 186, **245**, 367, 979, 1161

<array primary> • **300**

<array type> • **173**, 177, 182, 672, 705, 797, *1046*, 1048, 1049, *1054*, 1057, *1061*, 1063, *1067*, 1069, *1073*, 1075, *1078*, 1080, *1084*, 1086, 1161, 1162, 1165, 1187, 1197

<array value constructor> • *185*, 186, **302**, 303, 1211

<array value constructor by enumeration> • **302**, 303, 1162

<array value constructor by query> • 248, **302**, 303, 1162, 1211

<array value expression> • *245*, *247*, 249, **300**, 301, 1161

<array value expression 1> • **300**

AS • ?, ?, ?, ?, 94, *145*, 193, 205, 206, *212*, 214, 215, 216, 217, 218, 222, 223, 224, *230*, *232*, 233, 246, 288, 289, 291, 292, 293, 304, 305, 307, 308, *321*, 322, 323, 324, 332, 346, 347, *351*, *361*, 363, 364, 365, 366, *371*, 382, 388, 401, 446, 451, *504*, 521, 543, 544, *560*, 562, 569, *570*, *613*, 614, 619, *625*, *635*, *653*, *658*, *659*, 661, 663, 664, 665, 670, 672, 675, 681, 682, 683, 684, 697, 698, *699*, 706, 728, *729*, 730, *731*, *749*, 809, 812, 840, *852*, *854*, *862*, 867, *869*, *871*, 876, 920, *935*, *943*, 999, 1003, 1034, 1035, 1036, 1038, *1046*, 1048, 1049, *1053*, *1054*, 1056, 1057, 1058, *1061*, 1063, 1064, *1067*, 1068, 1069, *1073*, 1074, 1075, *1077*, *1078*, 1079, 1080, *1084*, 1085, 1086, 1096, 1104, 1105, 1109, 1156, 1179

<as clause> • **361**, 365, 366, 840, 1212, 1224

<as subquery clause> • ?, 227, *559*, **560**, 562, 563, 566, 568, 569, 1179

ASC • 60, 61, *144*, *546*

ASENSITIVE • 92, 98, 100, *145*, *835*, 836, 888, 1013, 1153, 1181

ASSERTION • *144*, 557, 602, 608, 612, 624, *648*, *650*, 651, 732, 737, 784

<assertion definition> • 103, 200, 327, *554*, **648**, 649, *815*, 1100, 1145, 1148

<assigned row> • 777, 778, 779, 781, 782, **875**, 876, 879, 1153, 1157

ASSIGNMENT • *144*, 670, *729*, 730

<asterisk> • 23, 76, **138**, 147, *251*, 252, *290*, 341, *361*, 363, 364, *411*, 413, 414, 423, *535*, 980, *1066*, 1068, 1186

<asterisked identifier> • **361**, 362

<asterisked identifier chain> • 341, **361**, 362, 370, 1183

ASYMMETRIC • *145*, *402*, 1186

AT • *145*, *285*, 293

ATOMIC • *145*, *653*, 1038

attempt to assign to non-updatable column • 1022, 1114

attempt to assign to ordering column • 1021, 1114

attempt to replace a zero-length string • 483, 1115

attempt to return too many result sets • 524, 1121
ATTRIBUTE • *144*, *677*, *679*
<attribute default> • 53, **674**, 675, 1156
<attribute definition> • 575, *658*, 661, 671, **674**, 675, *677*, 678, 1149, 1155, 1158
<attribute name> • **161**, 167, 168, *240*, 564, 617, 618, *659*, 661, 662, *674*, *679*, 1155
<attribute or method reference> • *185*, 186, **238**, 239, 1158
ATTRIBUTES • *144*, *977*
<attributes specification> • **977**, 987, 1133
<attributes variable> • 10, 838, 886, **977**, 978, 987
AUTHORIZATION • *145*, *553*, 554, *789*, *945*, *1030*, 1202
<authorization identifier> • 79, 117, 118, 119, **159**, 166, 200, 214, 215, 227, 241, 243, 277, 327, 505, 528, 532, 548, *553*, 554, 555, 556, 561, 567, 568, 571, 593, 594, 605, 611, 617, 618, 623, 626, 628, 633, 635, 636, 637, 639, 641, 644, 646, 648, 651, 655, 657, 668, 676, 697, 716, 717, 719, 720, 725, 728, 729, 730, 732, 735, 736, 739, 741, 743, 747, 750, 751, 752, *761*, 762, 765, 768, 774, *789*, 796, 854, 859, 864, 870, 871, 945, 1021, 1028, 1193, 1195, 1200, 1202
AVG • 63, *145*, *535*, 537, 540, 1197, 1201

— B —

Feature B011, “Embedded Ada” • 1050, 1123, 1129
Feature B012, “Embedded C” • 1059, 1123, 1129
Feature B013, “Embedded COBOL” • 1065, 1123, 1129
Feature B014, “Embedded Fortran” • 1070, 1123, 1129
Feature B015, “Embedded MUMPS” • 1076, 1123, 1129
Feature B016, “Embedded Pascal” • 1081, 1123, 1129, 1130
Feature B017, “Embedded PL/I” • 1088, 1123, 1130
Feature B021, “Direct SQL” • 1092, 1130
Feature B031, “Basic dynamic SQL” • 169, 191, 968, 969, 972, 976, 987, 996, 1000, 1005, 1007, 1008, 1010, 1015, 1016, 1017, 1018, 1020, 1022, 1130, 1131, 1132
Feature B032, “Extended dynamic SQL” • 169, 968, 990, 996, 1007, 1013, 1024, 1028, 1132
Feature B033, “Untyped SQL-invoked function arguments” • 526, 1132
Feature B034, “Dynamic specification of cursor attributes” • 987, 1132, 1133
Feature B041, “Extensions to embedded SQL exception declarations” • 1044, 1133
Feature B051, “Enhanced execution rights” • 793, 1040, 1133
Feature B111, “Module language Ada” • 793, 1123, 1133
Feature B112, “Module language C” • 793, 1123, 1133
Feature B113, “Module language COBOL” • 793, 1123, 1133
Feature B114, “Module language Fortran” • 793, 1123, 1133
Feature B115, “Module language MUMPS” • 793, 1123, 1133
Feature B116, “Module language Pascal” • 793, 1123, 1134
Feature B117, “Module language PL/I” • 793, 1123, 1134
Feature B121, “Routine language Ada” • 722, 1123, 1134
Feature B122, “Routine language C” • 722, 1123, 1134
Feature B123, “Routine language COBOL” • 722, 1123, 1134
Feature B124, “Routine language Fortran” • 723, 1123, 1134
Feature B125, “Routine language MUMPS” • 723, 1123, 1134
Feature B126, “Routine language Pascal” • 723, 1123, 1134
Feature B127, “Routine language PL/I” • 723, 1123, 1134, 1135
Feature B128, “Routine language SQL” • 723, 1124, 1135
<basic identifier chain> • **195**, 197, 198, *199*, *201*
<basic sequence generator option> • *603*, **749**, *751*
BEFORE • ?, ?, ?, ?, ?, ?, 71, 131, 132, 135, *144*, 197, *653*, 655, 914, 917
BEGIN • *145*, *653*, *1030*, 1038
BERNOULLI • *144*, *321*, 328
BETWEEN • *145*, 205, *352*, 353, *402*, 1185, 1186
<between predicate> • 248, 298, 367, *393*, 394, **402**, 473, 983, 1186, 1225
<between predicate part 2> • *208*, **402**
BIGINT • 14, 15, 30, *145*, *172*, 176, 182, 458, 462, 799, 807, 825, 827, 828, 829, 830, 958, 960, 1049, 1050, 1058, 1059, 1064, 1087, 1177, 1196
BIGINT • 824
BINARY • 14, 15, 28, *145*, *172*, 174, 175, 179, 226, 443, 457, 462, 810, 811, 813, 822, 823, 824, 825, 826, 827, 828, 829, 830, 958, 959, 963, 964, 975, 1005, *1046*, 1047, *1053*, 1056, 1058, *1061*, 1062, 1063, 1064, *1066*, 1068, *1077*, 1079, *1083*, *1084*, 1085, 1086, 1087, 1174
<binary concatenation> • 29, **265**, 266, 268, 1246
<binary factor> • **265**, 266, 268
<binary large object string type> • **172**, 174, 182, 1170, 1172
<binary overlay function> • 29, **272**, 276, 277, 283, 284, 1182
<binary position expression> • *253*, **254**, 258
<binary primary> • **265**, 266, 267
<binary set function> • 202, **535**, 538, 544, 545, 1143, 1144
<binary set function type> • **535**, 541, 545, 1190, 1202
<binary string literal> • *141*, 146, *150*, **151**, 153, 154, 155, 158, 576, 1170, 1172, 1195, 1219, 1245

<binary string type> • *171*, **172**, 175, 182, 995, 1170
<binary substring function> • 29, **272**, 276, 283, 1245
<binary trim function> • 29, **272**, 276, 277, 283, 284, 1245
<binary trim operands> • **272**
<binary trim source> • **272**, 277
<binary value expression> • *254*, 258, **265**, 266, 268, *272*, 276, 277, 283, *405*, 406, 467
<binary value function> • *270*, **271**, 272, 276, 277, 283, 284, 1171, 1175
BLOB • *145*, *172*, 174, 462, 1034, 1036, *1046*, 1047, 1048, *1054*, 1055, 1056, 1058, *1061*, 1062, 1063, *1067*, 1068, 1069, *1073*, 1074, *1077*, *1078*, 1079, *1084*, 1085, 1086
BOOLEAN • ?, ?, ?, ?, ?, ?, ?, ?, 14, 33, 97, *145*, 158, *173*, 177, 180, 181, 249, 299, 313, 459, 544, 574, 580, 581, 585, 735, 799, 809, 812, 824, 825, 827, 828, 829, 831, 958, 960, 981, 1049, 1058, 1070, *1077*, 1081, 1172, 1251
<boolean factor> • 71, **296**, 298
<boolean literal> • *150*, **153**, 158, 576, 1172
<boolean predicand> • 71, 211, **296**, 297, *312*, 313, 1138, 1172
<boolean primary> • **296**, 297, 298, 299, 981, 1172
<boolean term> • 71, **296**
<boolean test> • 71, **296**, 297, 299, 1147
<boolean type> • *171*, **173**, 180, 181, 1172
<boolean value expression> • 33, 52, 66, 71, *247*, 249, **296**, 297, 298, *311*, 312, 313, 434, 441, 592, 648, 1172
BOTH • *145*, 221, 222, 223, 224, 225, *271*, 275, 277, 282, 283, 935, 936, 945, 946, 948, 949, 951, 952, 953, 954, 967, 986, 1011
<bracketed comment> • **143**, 147, 149, 1184
<bracketed comment contents> • **143**, 147
<bracketed comment introducer> • **143**, 147
<bracketed comment terminator> • **143**
branch transaction already active • 923, 1119
BREADTH • *144*, *385*, 386
BY • ?, 59, 99, *145*, *330*, *340*, 341, 342, 345, 346, 347, 348, 349, *351*, 366, *371*, *385*, 473, *536*, 543, 544, 562, *570*, 573, 671, *733*, 735, *749*, *758*, 759, *766*, *769*, 770, *837*, 839, 1168, 1185

— C —

C • *144*, 476, *501*, 517, 518, 715, 722, 807, 809, 810, 811, 812, 813, 994, 1034, 1134
<C array locator variable> • *1053*, **1054**, 1057, 1059, 1165
<C array specification> • **1052**, *1053*, 1055, 1058
<C BINARY variable> • **1053**, 1056, 1059, 1171
<C BLOB locator variable> • *1053*, **1054**, 1056, 1059, 1173
<C BLOB variable> • *1053*, **1054**, 1056, 1059, 1173
<C character type> • **1052**, 1055
<C character variable> • **1052**, 1055, 1058, 1207
<C class modifier> • **1052**
<C CLOB locator variable> • *1053*, **1054**, 1057, 1059, 1173
<C CLOB variable> • **1053**, 1055, 1056, 1058, 1059, 1173, 1207
<C derived variable> • **1052**
<C host identifier> • *1031*, **1052**, *1053*, *1054*, 1055, 1056, 1057, 1058
<C initial value> • *1052*, *1053*, **1054**, 1058
<C multiset locator variable> • *1053*, **1054**, 1057, 1059, 1166
<C NCHAR variable> • **1053**, 1055, 1058
<C NCHAR VARYING variable> • **1053**, 1055, 1058
<C NCLOB variable> • **1053**, 1055, 1058
<C numeric variable> • **1052**, 1059, 1177
<C REF variable> • *1053*, **1054**, 1057, 1058, 1059, 1159
<C storage class> • **1052**
<C user-defined type locator variable> • *1053*, **1054**, 1057, 1059, 1164
<C user-defined type variable> • **1053**, 1056, 1059, 1167
<C VARBINARY variable> • **1053**, 1056, 1059, 1171
<C VARCHAR variable> • **1053**, 1055, 1058, 1207
<C variable definition> • *1030*, **1052**, 1055, 1057, 1058
<C variable specification> • **1052**
CALL • *145*, *919*
<call statement> • 86, 87, 91, 106, 110, 111, 112, 113, 115, 460, 504, 505, 507, 526, *816*, **919**, 966, 985, 995, 1100, 1132, 1206, 1236
CALLED • *145*, 661, 664, 682, *701*, 704
CARDINALITY • *145*, *254*, 323, 434, 436, 438, 959, 960, *970*, 974, 995
<cardinality expression> • 31, *253*, **254**, 257, 259, 263, 1161, 1169
cardinality violation • 391, 849, 866, 1017, 1114
CASCADE • 132, *144*, *556*, 557, *583*, 585, 586, 587, 588, 589, 601, 602, 604, 605, 608, 609, 611, 612, 622, 623, 624, 634, 638, 642, 646, 651, 697, 698, 727, 728, 731, 732, 736, 737, 745, 747, 748, 752, 783, 784, 785, 881, 1135, 1251, 1252
CASCADED • 56, 59, *145*, *613*, 614, 615, 619, 905, 912
CASE • *145*, 205, *208*, 209, 211, 304, 305, 307, 387, 1110, 1138
<case abbreviation> • **208**, 209, 981, 1235
<case expression> • *185*, 186, **208**, 211, 227, 367, 453, 1235
<case operand> • **208**, 209, 211, 982, 1138
<case specification> • **208**, 209, 210, 211, 982
CAST • 50, *145*, 193, 205, *212*, 214, 215, 216, 217, 218, 222, 223, 224, 288, 289, 291, 292, 293, 323, 346, 347, 366, 382, 401, 446, 451, 521, 544, *659*, 665, 670, 672, 675, 682, 697, 698, *700*, 706, 728, *729*, *731*, 809, 812, 876, 920, 999, 1003, 1035, 1038, 1096, 1109, 1233

<cast function> • **729**, 730
<cast operand> • **212**, 213, 215, 226, 248, 981, 1143, 1159, 1174
<cast specification> • 18, 34, 35, 69, 96, *185*, 186, **212**, 213, 214, 215, 216, 217, 218, 226, 248, 617, 731, 981, 999, 1003, 1234
<cast target> • **212**, 226, 731, 981, 1159
<cast to distinct> • *658*, **659**, 660, 661, 673, 1158
<cast to distinct identifier> • **659**, 661
<cast to ref> • *658*, **659**, 660, 661, 662, 673, 1158
<cast to ref identifier> • **659**, 662
<cast to source> • *658*, **659**, 660, 661, 673, 1158
<cast to source identifier> • **659**, 661
<cast to type> • *658*, **659**, 660, 661, 662, 673, 1158
<cast to type identifier> • **659**, 662
CATALOG • *144*, *948*, 949
<catalog name> • 79, **159**, 165, 166, 167, 168, 190, 283, 554, 556, 790, 948, 949, 1106, 1107, 1108, 1109, 1147, 1195, 1204
<catalog name characteristic> • **948**, 984
CATALOG_NAME • *144*, *1096*, 1106, 1107, 1108
CEIL • *145*, *255*
CEILING • *145*, *255*
<ceiling function> • 32, *253*, **255**, 257, 261, 264, 1189, 1198
CHAIN • ?, ?, ?, ?, 126, *144*, *931*, 932, *933*, 934, 1181
CHAR • ?, ?, ?, *145*, *171*, *172*, 174, 462, 799, 805, 806, 808, 1047, 1049, *1077*, 1078, 1079, 1080, 1081, *1083*
<char length expression> • **254**, 258, 259, 979, 1223, 1245
<char length units> • **172**, 174, 177, 182, *253*, *254*, 256, 258, 259, 262, 264, *270*, *271*, 273, 274, 278, 281, 284, 1154, 1176
CHAR_LENGTH • *145*, *254*, 258, 275
CHARACTER • 14, 15, 17, 18, 96, *145*, 155, *171*, *172*, 174, 175, 179, 183, 226, 263, 410, 416, 443, 457, 462, 527, *553*, 557, 571, *635*, *637*, 638, *761*, 805, 806, 808, 809, 810, 811, 812, 813, 822, 823, 825, 826, 827, 828, 829, 830, 958, 959, 964, 975, 979, 980, 984, 985, 1005, *1045*, *1046*, 1047, 1049, *1052*, *1053*, 1055, 1058, *1060*, *1061*, 1062, 1064, *1066*, 1067, 1068, 1070, *1072*, 1073, *1077*, 1078, 1081, *1083*, 1084, 1085, 1087, 1109, 1143, 1174, 1175, 1191, 1195, 1196
<character enumeration> • **412**, 414, 415
<character enumeration exclude> • **412**, 415
<character enumeration include> • **412**, 415
<character factor> • **265**, 266, 267, 280, 1198
<character large object type> • **171**, 174, 182, 1172
<character like predicate> • **405**, 406, 409, 984, 1138, 1171
<character like predicate part 2> • *208*, **405**
character not in repertoire • 177, 1115
<character overlay function> • 21, 29, *270*, **271**, 272, 275, 278, 284, 1182
<character pattern> • **405**, 409, 467, 984, 1139
<character position expression> • *253*, **254**, 255, 258
<character primary> • **265**, 266, 267
<character representation> • **150**, 153, 154, 155, 156, 158, *1045*, 1049, *1054*, 1058, *1060*, 1064, *1083*, 1087, 1138, 1223
<character set definition> • 104, 164, 527, *554*, **635**, 636, *815*, 1100, 1144
<character set name> • 154, **161**, 164, 165, 167, 169, 174, *527*, 557, 626, *635*, 636, *637*, 643, 644, 754, *761*, 762, 772, 805, 951, 1144, 1195
<character set name characteristic> • **951**, 984
<character set source> • **635**
<character set specification> • 26, 148, *150*, 154, 155, 162, *171*, 175, 183, **527**, 528, *553*, 555, 571, 625, *635*, 636, *639*, *643*, 674, 754, 784, *790*, 791, *794*, 954, 959, *1030*, *1045*, *1046*, 1049, *1052*, *1053*, 1058, *1060*, *1061*, 1064, *1066*, 1070, *1072*, *1077*, 1081, *1083*, 1087, 1145, 1202, 1206
<character set specification list> • **789**, 791, *954*, 955
<character specifier> • **411**, *412*, 413, 414
<character string literal> • *142*, 146, 148, **150**, 153, 154, 155, 156, 158, *160*, 162, 575, 576, 951, 1138
<character string type> • **171**, 174, 175, 625, 674, 994, 1196, 1222, 1223
<character substring function> • 20, 29, **270**, 272, 277, 278, 1223, 1245
<character transliteration> • 22, **270**, 272, 274, 278, 281, 284, 1149, 1198
<character value expression> • *253*, *254*, 258, **265**, 266, 267, 268, *270*, *271*, 272, 273, 274, 275, 276, 278, 279, 280, 281, 282, *405*, 410, *411*, 412, 413, 414, 416, 1143, 1175, 1198
<character value expression 1> • **254**, 255, 258
<character value expression 2> • **254**, 255, 258
<character value function> • **270**, 272, 277
CHARACTER_LENGTH • 8, *145*, *254*, 262, 278, 281, 464
CHARACTER_SET_CATALOG • *144*, 958, 959, *970*, 974, 975, 976, 994, 1206
CHARACTER_SET_NAME • *144*, 958, 959, *970*, 974, 975, 976, 994, 1206
CHARACTER_SET_SCHEMA • *144*, 958, 959, *970*, 974, 975, 976, 994, 1206
CHARACTERISTICS • ?, ?, ?, *144*, *943*
CHARACTERS • *144*, *172*, 174, 177, 256, 258, 273, 274
CHECK • 56, 59, *145*, 572, *592*, 593, *613*, 614, 615, 619, 621, 634, *648*, 861, 866, 868, 873, 895, 905, 912, 1107, 1148, 1151, 1177
<check constraint definition> • 200, 226, 327, *570*, 572, *579*, 580, **592**, 593, *625*, 1148, 1229

<circumflex> • 23, *138*, **139**, *412*, 413, 415
CLASS_ORIGIN • *144*, *1096*, 1106, 1122, 1207
CLOB • *145*, *172*, 174, 462, 1034, 1036, *1046*, 1047, 1048, *1053*, *1054*, 1055, 1056, 1058, *1060*, *1061*, 1062, 1063, *1066*, *1067*, 1068, 1069, *1072*, *1073*, 1074, *1077*, 1078, 1079, *1083*, *1084*, 1085, 1086
CLOSE • *145*, *847*, *1018*
<close statement> • 99, 105, 109, 112, 114, 115, *816*, **847**, 1100, 1228
CLU • 262
COALESCE • 8, 73, *145*, *208*, 209, 332, 336
COBOL • ?, ?, ?, 97, *144*, 476, *501*, 517, 518, 715, 722, 725, 793, 807, 808, 809, 810, 811, 812, 813, 1033, 1034, 1061, 1062, 1063, 1064, 1065, 1123, 1129, 1133, 1134, 1207
<COBOL array locator variable> • *1060*, **1061**, 1063, 1065, 1165
<COBOL binary integer> • **1061**, 1064
<COBOL BINARY variable> • *1060*, **1061**, 1062, 1065, 1171
<COBOL BLOB locator variable> • *1060*, **1061**, 1063, 1065, 1173
<COBOL BLOB variable> • *1060*, **1061**, 1062, 1065, 1173
<COBOL character type> • **1060**, 1062, 1064
<COBOL CLOB locator variable> • *1060*, **1061**, 1063, 1065, 1173
<COBOL CLOB variable> • **1060**, 1062, 1065, 1173
<COBOL derived type specification> • **1060**
<COBOL host identifier> • *1031*, **1060**, 1062, 1064
<COBOL integer type> • *1060*, **1061**
<COBOL multiset locator variable> • *1060*, **1061**, 1064, 1065, 1166
<COBOL national character type> • **1060**, 1061, 1062
<COBOL NCLOB variable> • **1060**, 1062
<COBOL nines> • **1061**
<COBOL nines specification> • **1061**
<COBOL numeric type> • *1060*, **1061**, 1064
<COBOL REF variable> • *1060*, **1061**, 1064, 1065, 1159
<COBOL type specification> • **1060**
<COBOL user-defined type locator variable> • *1060*, **1061**, 1064, 1065, 1164
<COBOL user-defined type variable> • *1060*, **1061**, 1063, 1065, 1167
<COBOL variable definition> • *1030*, **1060**, 1062, 1064, 1207
COLLATE • *145*, *532*
<collate clause> • 61, *171*, 174, 212, *265*, 266, *340*, 341, *351*, 352, 354, **532**, *570*, 571, 574, *625*, 627, *635*, 636, *674*, 675, 1148, 1149
COLLATION • *144*, *188*, 557, *639*, *641*, 642, 754, *761*, 785, *789*, *954*
<collation definition> • 104, 164, *554*, **639**, 640, 772, *815*, 1100, 1148, 1202, 1213
<collation name> • 61, **160**, 164, 167, 168, 341, 352, *532*, 557, 626, 635, 636, *639*, *641*, 754, *761*, 762, 772, *789*, 791, 954, 1148
<collation specification> • **954**
COLLATION_CATALOG • *144*, *970*, 973, 994, 1206
COLLATION_NAME • *144*, *970*, 973, 994, 1206
COLLATION_SCHEMA • *144*, *970*, 973, 994, 1206
COLLECT • *145*, *535*, 537, 545, 1169
<collection derived table> • 74, *321*, **322**, 325, 326, 329, 1162, 1169
<collection type> • 47, 48, *171*, **173**, 177, 181, 182, 1170
<collection value constructor> • **185**, 186
<collection value expression> • **247**, 249, *254*, 257, 259, *322*, 453
<colon> • *138*, **139**, *152*, *160*, *412*, 413, 415, 499, 500, *1031*, *1045*, *1077*
COLUMN • *145*, *595*, *597*, *604*, 605, 784, 785, 1135
<column constraint> • 562, **570**, 572, 1229
<column constraint definition> • ?, *560*, 562, 564, 565, **570**, 572, 573
<column default option> • **560**, 561
<column definition> • ?, 68, 175, 182, 555, *559*, 561, 563, 564, 565, 566, 568, **570**, 571, 573, 574, 575, 582, *595*, 596, 880, 881, 1149, 1159, 1188, 1230
<column name> • 57, **160**, 166, 195, 196, 197, *199*, *322*, 323, 326, 332, *361*, 363, 365, 366, 372, 377, 378, *385*, 386, 387, 388, *560*, 563, 564, 565, *570*, 571, 581, 582, 584, 595, 596, *597*, 598, 599, 600, 601, 603, *604*, *613*, 614, 617, 618, 619, 654, *700*, 702, 754, 762, 763, 771, 772, 773, 774, 777, 778, 781, 784, 785, 840, 841, 858, 859, 860, 863, 864, 867, 870, *875*, 1022, 1028, 1038, 1107, 1188, 1212
<column name list> • 118, 119, **322**, *331*, *361*, *371*, *385*, 386, *560*, 563, *581*, *583*, *613*, *653*, *761*, *837*, 839, 840, 841, *857*, 870, 1022, 1028, 1219, 1229, 1230
<column option list> • **560**, 565, 568, 1159
<column options> • *559*, **560**, 564, 565, 566
<column reference> • 60, 62, 135, *185*, 186, *187*, *188*, 189, 190, 197, **199**, 200, *202*, 203, 297, *330*, 339, *340*, 345, 346, 349, *351*, 352, 364, 365, 369, 508, 523, 562, 571, 605, 774, 775, 776, 777, 778, 779, 781, 839, 840, 843, 848, 869, 1028, 1143, 1153, 1182, 1185, 1210, 1228, 1236
COLUMN_NAME • *144*, *1096*, 1107, 1108
<comma> • ?, ?, **138**, *172*, *173*, 190, *202*, *208*, *254*, *255*, *271*, *302*, *308*, *311*, *316*, *319*, *322*, *330*, *340*, 342, 343, 344, 345, *351*, *361*, 365, *371*, *385*, *403*, *411*, *439*, *497*, *503*, *504*, *529*, *535*, *536*, *546*, *559*, *613*, *658*, *659*, *699*, *700*, *701*, *738*, *741*, *744*, *758*, *761*, *766*, *769*, *790*, *795*, *843*, *848*, *862*, *875*, *882*, *883*, *925*, *927*, *943*, *970*, *973*,

997, *1001*, *1045*, *1052*, *1053*, *1054*, *1066*, *1072*, *1077*, *1083*, *1095*, *1096*
COMMAND_FUNCTION • *144*, *1095*, 1099, 1109
COMMAND_FUNCTION_CODE • *144*, *1095*, 1099, 1207
<comment> • **143**, 147
<comment character> • **143**
COMMIT • 56, 123, 125, *145*, *559*, 565, 567, 584, 592, *880*, *931*
<commit statement> • ?, ?, 67, 92, 99, 103, 106, 108, 110, 111, 121, 124, 126, 792, *816*, **931**, 932, 940, 1090, 1100, 1181, 1215, 1230
COMMITTED • 122, 124, 125, *144*, 924, *925*
<common sequence generator option> • **749**
<common sequence generator options> • 492, *570*, 572, **749**, 750
<common value expression> • 71, 211, **247**, 249, 297, *311*, *312*, 313, 405, 409, 412, 417, 425, 434, 436, 438, 439, 1138, 1171
<comp op> • 7, 71, **395**, 396, 397, 398, 399, 400, 401, *421*, 422, 547, 1212, 1232
<comparison predicate> • 7, 23, 29, 71, 188, 248, 298, 367, *393*, 394, **395**, 396, 399, 421, 422, 466, 471, 473, 547, 983, 1232, 1234
<comparison predicate part 2> • *208*, **395**
<computational operation> • **535**, 544, 545, 1143, 1169, 1170, 1172, 1176, 1189, 1227
<concatenation> • **265**, 266, 267, 268, 1223, 1246
<concatenation operator> • 19, *142*, **143**, *265*, *300*, 980
CONDITION • *145*, *1095*, *1096*, 1110, 1219
<condition> • **1041**, 1042
<condition action> • **1041**, 1043
<condition information> • **1095**, 1106, 1110
<condition information item> • *1095*, **1096**, 1106, 1110
<condition information item name> • **1096**, 1097, 1109, 1110, 1111, 1137
<condition number> • *1095*, **1096**, 1097, 1106, 1110, 1215
CONDITION_NUMBER • *144*, *1096*, 1106, 1109, 1110
CONNECT • *145*, 819, *935*, 1091
<connect statement> • 106, 116, 126, 128, 135, *816*, 819, **935**, 936, 937, 1090, 1100, 1152, 1194, 1205, 1214
CONNECTION • *144*, 819, *938*, 1091
connection does not exist • 819, 938, 940, 1091, 1114
connection exception • 126, 819, 936, 938, 940, 1091, 1114
connection failure • 938, 1114
<connection name> • 126, **161**, 166, 167, 168, *935*, 936, *938*, 940, 1109, 1152
connection name in use • 936, 1114
<connection object> • **938**, 939, *940*
<connection target> • **935**
<connection user name> • 128, **161**, 166, *935*, 936, 937, 1205
CONNECTION_NAME • *144*, *1096*, 1109
CONSTRAINT • *145*, *533*, 602, *607*, 608, 612, *632*, 651, 737, 784, *1041*, 1042, 1043, 1044, 1133, 1251, 1252
<constraint characteristics> • **533**, 534, *570*, 572, *579*, *625*, 634, *648*, 649, 1150
<constraint check time> • **533**
<constraint name> • **160**, 164, 165, 167, 169, *533*, 557, 579, 602, *607*, 608, 625, 626, *632*, 634, *648*, 649, *650*, 651, 732, 737, 784, *927*, *1041*, 1043, 1044, 1145, 1213
<constraint name definition> • **533**, 534, *570*, 572, *579*, *625*, 626, 634, 1145, 1150, 1213
<constraint name list> • **927**
CONSTRAINT_CATALOG • *144*, *1096*, 1106, 1108
CONSTRAINT_NAME • *144*, *1096*, 1106, 1108
CONSTRAINT_SCHEMA • *144*, *1096*, 1106, 1108
CONSTRAINTS • ?, *144*, *927*, 931, 1252
CONSTRUCTOR • 40, 42, *144*, *529*, 530, 531, *660*, 663, 664, 665, 666, 667, 669, 681, 683, 684, 685, 687, 688, 690, *692*, 693, 694, 695, *700*, 702, 703, 719
<constructor method selection> • **232**, 233, 505, 506, 507
containing SQL not permitted • 518
containing SQL not permitted • 1118
CONTAINS • *144*, 661, 682, *701*, 709, 716, 720, 726, 735, 1203
<contextually typed row value constructor> • **311**, 312, 313, *314*, 315, 317, 858, 863, 1147, 1176, 1211
<contextually typed row value constructor element> • **311**, 312, 313, 858, 863
<contextually typed row value constructor element list> • **311**, 313, 317, 1147
<contextually typed row value expression> • 228, **314**, 315, *316*, 317, 858, 859, 863, *875*, 982, 984, 1227
<contextually typed row value expression list> • **316**, 317, 1147
<contextually typed table value constructor> • 228, 313, **316**, 317, *857*, 858, 859, 861, 863, 868, 982, 1147, 1157, 1211, 1227
<contextually typed value specification> • **193**, *311*, 312, 859, *862*, 864, *875*, 877, 1234
CONTINUE • *144*, *1041*
CONVERT • *145*, *270*
CORR • 64, *145*, *535*, 541
<correlation name> • ?, ?, ?, 73, **160**, 166, 196, *321*, 322, 324, 325, 362, 363, 386, *654*, *852*, *854*, 855, *862*, 863, 865, 867, 868, *869*, *871*, 872, 1105, 1152, 1225, 1226
CORRESPONDING • *145*, *371*, 377, 384, 604, 621, 1139
<corresponding column list> • **371**, 377
<corresponding spec> • **371**

COUNT • 63, *145*, 205, 366, 367, *535*, 537, 539, 540, 544, *970*, 971, 992, 998, 1002, 1104, 1105, 1143, 1197, 1201, 1252

COVAR_POP • 64, *145*, *535*, 541

COVAR_SAMP • 64, *145*, *535*, 541

CREATE • ?, ?, ?, ?, ?, *145*, *553*, *559*, *613*, 614, *625*, *635*, *639*, *643*, *648*, *653*, *658*, 670, 671, 672, *699*, *729*, *733*, 735, *738*, *749*, *765*, 1251, 1252

CROSS • 72, *145*, *330*, 333

<cross join> • **330**, 333, 336, 337, 1141

CUBE • 62, 70, 76, *145*, *340*, 343

<cube list> • **340**, 341, 342, 343, 344, 348, 1185

CUME_DIST • 62, 63, 65, *145*, *204*, 205, 207, 544, 1188, 1193

CURRENT • *145*, 205, *352*, 353, 356, 358, 359, *852*, *869*, *940*, *1019*, *1021*, *1023*, *1027*

<current collation specification> • *187*, **188**, 189, 190, 192, 1149

<current date value function> • **288**, 1233

<current local time value function> • **288**, 289, 1146, 1233

<current local timestamp value function> • **288**, 289, 1146, 1233

<current time value function> • **288**, 289, 1142

<current timestamp value function> • **288**, 289, 1142

CURRENT_CATALOG • 128, *145*, *187*, 188, 190, 191, 248, *575*, 576, 577, 578, 885, 1093, 1151, 1197, 1219

CURRENT_DATE • 124, *145*, 224, *288*, 289, 298, 593, 649, 1148, 1211

CURRENT_DEFAULT_TRANSFORM_GROUP • 129, *145*, *187*, 189, 191, 885, 1093, 1167

CURRENT_PATH • 93, 128, *145*, *187*, 188, 190, 191, 248, *575*, 576, 577, 578, 885, 1093, 1160, 1197

CURRENT_ROLE • ?, ?, 116, *145*, *187*, 188, 190, 191, 248, 367, *575*, 576, 577, 578, *762*, 787, 885, 1093, 1184

CURRENT_SCHEMA • 128, *145*, *187*, 188, 190, 191, 248, *575*, 576, 577, 578, 885, 1093, 1152, 1197, 1219

CURRENT_TIME • *145*, *288*, 289, 1211

CURRENT_TIMESTAMP • *145*, *288*, 289, 298, 593, 649, 720, 1148, 1211

CURRENT_TRANSFORM_GROUP_FOR_TYPE • 129, *145*, *187*, 189, 191, 885, 886, 1093, 1167

CURRENT_USER • 116, 124, *145*, *187*, 188, 190, 191, 248, 367, *575*, 576, 577, 578, *761*, 787, 885, 1093, 1139, 1140, 1197

CURSOR • ?, ?, *145*, *835*, 838, *991*

<cursor attribute> • **988**

<cursor attributes> • 100, **988**

<cursor holdability> • 100, **835**, 836, 838, 886, *988*, 1187, 1228

<cursor intent> • 838, **1011**

<cursor name> • ?, 97, 98, **160**, *161*, 162, 163, 167, 169, 791, *833*, 834, 838, *842*, *843*, *847*, *852*, *869*, *1009*, 1014, 1016, 1018, 1019, 1021, 1022, *1025*, 1032, 1033, 1188, 1234

cursor operation conflict • 590, 856, 866, 873, 893, 895, 1121

cursor operation conflict • 1106

<cursor properties> • 100, *833*, 834, **835**, 837, 838, *1009*, *1011*, 1012, 1186

<cursor returnability> • 100, **835**, 836, 838, 886, *988*, 1186

<cursor scrollability> • 100, **835**, 836, 838, 841, 886, *988*, 1143, 1153

<cursor sensitivity> • 100, **835**, 836, 838, 886, *988*, 1153, 1181

cursor sensitivity exception • 855, 860, 865, 872, 887, 892, 894, 1114

<cursor specification> • 8, 56, 62, 84, 92, 98, 99, 100, 101, 195, 204, 205, 227, 353, 777, 778, 780, 781, 782, *833*, 834, **837**, 839, 843, 869, 870, 885, 886, 888, *978*, 986, 989, 1009, 1011, 1012, 1014, 1021, 1022, 1023, 1027, 1028, *1093*, 1153, 1177, 1210, 1213

cursor specification cannot be executed • 1006, 1117

CURSOR_NAME • *144*, *1096*, 1106, 1107

CYCLE • 81, *145*, *385*, 491, 493, *750*

<cycle clause> • **385**, 386, 387, 388

<cycle column> • **385**

<cycle column list> • **385**, 386

<cycle mark column> • **385**, 386

<cycle mark value> • **385**, 386

— D —

DATA • ?, *144*, *560*, 568, *701*, 704, 709, 716, 720, 726, 813, *970*, 971, 972, 973, 974, 993, 998, 999, 1005, 1047, 1062, 1068, 1074, 1079, 1085, 1203, 1214

data exception • 177, 190, 191, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 245, 252, 259, 260, 261, 262, 267, 268, 278, 279, 282, 283, 284, 287, 294, 301, 303, 316, 328, 355, 356, 357, 358, 359, 407, 408, 413, 444, 445, 446, 447, 449, 450, 451, 452, 477, 480, 483, 491, 510, 522, 523, 524, 539, 540, 543, 675, 797, 809, 845, 846, 850, 851, 878, 879, 947, 971, 972, 974, 1109, 1115, 1197, 1198

<data type> • ?, ?, 19, 34, 45, 46, 47, 83, 90, 91, 93, 155, 166, **171**, *173*, 174, 176, 177, 179, 180, 181, 182, *183*, *212*, 213, 226, *232*, 233, 312, 326, 333, 366, 368, 477, 478, 507, 520, 521, *529*, 530, 564, 568, *570*, 571, 573, 594, 618, 661, 665, 668, *674*, 675, 682, 689, *699*, *700*, *701*, 703, 705, 706, 712, 714, 715, 718, 720, 721, 722, 725, *729*, *749*, 750, 775, 776, 777, 778, 779, 781, 791, *795*, 797, 798, 806, 807, 818, 981, 1033, 1158, 1159, 1162, 1164, 1165, 1166, 1169, 1170, 1173, 1196, 1229, 1245

<data type list> • **529**, 530, *692*, 693

<data type or domain name> • 68, **570**, 571

data type transform function violation • 1000, 1004, 1117

DATE • 14, 15, 33, 35, 36, 96, *145*, *151*, 155, *173*, 176, 178, 180, 221, 222, 285, 288, 459, 462, 824, 826, 827, 828, 829, 831, 975, 985
<date literal> • **151**, 155, 1232
<date string> • *142*, **151**
<date value> • **152**, 157
<datetime factor> • 248, **285**
datetime field overflow • 287, 355, 446, 451, 1115
<datetime literal> • *150*, **151**, 156, 157, 576, 1232
<datetime primary> • 248, **285**, 286
<datetime term> • 248, **285**, 286, 287, *290*, 291, 293
<datetime type> • 34, *171*, **173**, 177, 178, 180, 958, 995, 1197, 1232
<datetime value> • *152*, **153**, 156
<datetime value expression> • *247*, 248, 249, *254*, 256, 257, 258, **285**, 286, 287, *290*, 291, 293, 1136, 1233
<datetime value function> • 248, *285*, 286, **288**, 289, *575*, 576, 577, 886, 1233
DATETIME_INTERVAL_CODE • *144*, 958, *970*, 974, 975, 976, 995
DATETIME_INTERVAL_PRECISION • *144*, 958, *970*, 974, 975, 976, 995
DAY • 35, *145*, 156, 178, 286, 355, 356, 454, 459, *497*, 498, 976, 981, 985
<day-time interval> • **152**
<day-time literal> • **152**, 156
<days value> • **152**, 156
DEALLOCATE • *145*, 519, 792, *969*, 986, *989*, 1008, 1090
<deallocate descriptor statement> • 84, 107, 109, 792, *817*, **969**, 1090, 1100, 1130
<deallocate prepared statement> • 83, 84, 98, 99, 107, 109, *817*, **989**, 990, 1100, 1132
DEC • *145*, *172*, 174, 179, 809, 812, *1072*, 1073, *1084*, 1196
DECIMAL • 14, 15, 30, *145*, *172*, 174, 175, 176, 180, 458, 462, 824, 825, 827, 828, 829, 830, 958, 959, 975, 1073, *1084*, 1087, 1196
DECLARE • *145*, *833*, *880*, *1009*, *1030*, 1038, *1083*, 1087
<declare cursor> • 10, 81, 97, 98, 100, 105, 110, 112, 114, 115, 163, *790*, 791, **833**, 834, 837, 838, 842, 1009, *1029*, 1032, 1034, 1039, 1213, 1215, 1228
DEFAULT • 59, 132, *145*, *193*, *385*, *553*, 562, *570*, 573, *575*, *583*, 585, 587, 589, 590, *599*, *630*, 819, *857*, 858, 861, 895, *935*, 936, *938*, 940, *953*, 1091, 1137, 1205, 1251
<default clause> • 51, *560*, 565, *570*, 573, **575**, 576, 578, *598*, *625*, 626, *629*, 633, *674*, 858, 1229, 1233
<default option> • 51, 52, 193, 562, 565, 573, **575**, 576, 577, 578, 737, 1139, 1151, 1152, 1160, 1184
<default specification> • **193**, 194, 313, 858, 860, 863, 864, 868, 876, 1234
default value too long for information schema • 578, 1121
DEFAULTS • *144*, *560*, 561, 562
DEFERRABLE • *144*, *533*, 534, 572, 579, 625, 648, 1150, 1251, 1252
DEFERRED • *144*, *533*, *927*, 1251, 1252
DEFINED • *144*, 514, *701*, 709, 721, 958, 960, 1201
DEFINER • ?, *144*, 514, *701*, 709, 718, 719, 720, 780, 986
DEGREE • *144*, 959, 960, *970*, 974, 975, 995
DELETE • 56, 118, 119, 120, 133, 134, 135, *145*, 548, 549, 550, 551, *559*, 565, 567, 568, *583*, 584, 592, *653*, 654, *761*, 770, 778, 781, 820, *852*, 853, *854*, 855, 880, 881, 896, 916, 931, *1019*, *1023*, 1024, 1251, 1252
<delete rule> • **583**, 585, 586, 587, 588, 590, 1137
<delete statement: positioned> • 58, 92, 100, 101, 105, 109, 112, 114, 115, 590, 778, 781, *816*, **852**, 853, 856, 866, 873, 893, 894, 1100, 1228
<delete statement: searched> • 58, 105, 109, 112, 113, 115, 777, 778, 779, 781, 782, *816*, **854**, 856, 893, 894, 931, *977*, *1089*, 1100, 1104, 1105, 1152, 1177, 1215, 1227
<delimited identifier> • **142**, 146, 147, 148, 149, *159*, 190, 1141, 1191, 1195, 1224
<delimited identifier body> • **142**, 147, 148, 149, 1141
<delimited identifier part> • **142**, 147, 149, 1141
<delimiter token> • *141*, **142**, 147
DENSE_RANK • 62, 65, *145*, *204*, 205, 538, 1202
dependent privilege descriptors still exist • 783, 1117
<dependent variable expression> • 64, 65, 202, *535*, **536**, 538, 541, 542, 544, 545, 1143, 1144
DEPTH • *144*, *385*, 386
DEREF • *145*, 241, *243*
<dereference operation> • 238, **240**, 367, 601, 610, 622, 775, 776, 778, 779, 781, 1158
<dereference operator> • **238**, *240*, *241*
DERIVED • *144*, *560*, 563, 616, 619
<derived column> • 195, 274, 332, 339, 345, **361**, 364, 365, 366, 367, 423, 839, 840, 846, 993, 1016, 1143, 1186, 1213
<derived column list> • *321*, **322**, 323, 326, 1225
<derived representation> • 16, 47, **659**, 662, 669
<derived table> • 60, 74, *321*, **322**, 326, 327, 328, 329, 367, 374, 378, 1147
DESC • 60, 61, *144*, 356, 357, 358, 544, *546*, 547
DESCRIBE • *145*, *991*
<describe input statement> • 84, 107, 109, **991**, 992, 993, 996, 998, 1132, 1214, 1237
<describe output statement> • 84, 107, 109, **991**, 992, 993, 996, 1002, 1131, 1214
<describe statement> • *817*, 972, 976, **991**, 1100, 1206
<described object> • **991**
DESCRIPTOR • *144*, 792, *967*, *969*, *970*, *973*, *991*, *1001*, 1090

<descriptor item name> • **970**, 971, 972, *973*, 974, 1182

<descriptor name> • **161**, 166, 169, 792, *967*, *969*, *970*, 971, 972, *973*, 974, *991*, 992, 997, 998, *1001*, 1002, 1090, 1130, 1132, 1206, 1234

DETERMINISTIC • 65, *145*, 517, 661, 664, 670, 671, 672, 682, *701*, 704, 709, 716, 718, 735, 1203

<deterministic characteristic> • *660*, 664, 682, *700*, **701**, 704

DIAGNOSTICS • ?, ?, *144*, 921, *925*, 926, *1095*

diagnostics exception • 1112, 1117

<diagnostics size> • ?, ?, ?, **925**, 926, 943, 944, 1137

<digit> • **137**, *141*, 147, *151*, 155, 158, *159*, 174, 218, 220, 415, 876, *1041*, 1113, 1146, 1207

<direct implementation-defined statement> • **1089**, 1090, 1091, 1207

<direct invocation> • **232**

<direct select statement: multiple rows> • 105, 109, 113, 114, 115, *1089*, 1090, **1093**, 1101

<direct SQL data statement> • **1089**, 1092

<direct SQL statement> • 85, 89, 93, 121, 126, 128, 131, 135, 162, 163, 164, 165, 475, 506, 507, 717, 818, 819, 948, 949, 951, 952, **1089**, 1090, 1091, 1092, 1130, 1193, 1215

<directly executable statement> • **1089**

DISCONNECT • *145*, *940*

disconnect error • 941, 1121

<disconnect object> • **940**

<disconnect statement> • 106, 126, *816*, **940**, 941, 1101, 1152, 1214

DISPATCH • *144*, 672, *700*, 735

<dispatch clause> • **700**, 705

DISPLAY • *1061*

DISTINCT • ?, ?, 23, 29, 50, 63, 64, *145*, 205, 206, 227, 249, *304*, 305, 306, 307, 348, 362, 365, 367, 369, 370, *371*, 375, 377, 380, 381, 382, 384, 400, *432*, 433, 469, 471, *535*, 537, 538, 539, 544, 545, *659*, 840, 1126, 1139, 1143, 1146, 1153, 1170, 1176, 1178, 1179, 1185, 1186, 1212

<distinct predicate> • 248, *393*, 394, **432**, 433, 466, 467, 983, 1178, 1179

<distinct predicate part 2> • *209*, **432**, 433, 1179

DISTINT • 1126

division by zero • 252, 259, 1115

DOMAIN • ?, ?, *144*, 557, *625*, *628*, *633*, 634, 732, 737, 754, *761*, 784

<domain constraint> • 51, 189, 226, **625**, 626, *631*

<domain definition> • 103, 165, 175, *553*, 555, 575, 592, **625**, 626, 627, 772, *815*, 1101, 1138, 1149, 1213

<domain name> • 51, 68, **159**, 165, 166, 169, 199, *212*, 214, 215, 226, 366, 557, *570*, 571, 572, 573, 574, 578, *625*, 626, *628*, 629, 630, 631, 632, *633*, 732, 737, 754, *761*, 762, 772, 784, 981, 1138

DOUBLE • 14, 15, 30, *145*, *172*, 176, 180, 458, 462, 824, 825, 827, 828, 829, 831, 958, 960, 1049, 1058, *1066*, 1070, 1196, 1197

<double colon> • **143**, *234*

<double period> • **143**, *1045*, *1077*

<double quote> • 20, *137*, **138**, *142*, 146, 147, 148, 279, 283

<doublequote symbol> • **142**, 148, 283

DROP • ?, *145*, *556*, 557, *599*, *601*, 602, *604*, 605, *607*, 608, 609, *610*, 611, 612, *622*, 623, 624, *630*, *632*, *633*, *637*, *641*, *646*, *650*, 651, *657*, *679*, *692*, *696*, 697, 698, *727*, 728, *731*, 732, *736*, 737, *744*, 745, *746*, 747, 748, *752*, *768*, 784, 785, 881, 1135

<drop assertion statement> • 103, 557, 602, 608, 612, 624, **650**, 651, 652, 732, 737, 784, *816*, 1101, 1145

<drop attribute definition> • *676*, **679**, 680

<drop behavior> • **556**, *601*, *604*, *607*, *610*, 612, *622*, 624, *633*, *641*, *650*, *696*, 698, *727*, 728, *731*, *736*, *744*, *746*, *752*, *769*, 785, 1135, 1231

<drop character set statement> • 104, 164, 557, **637**, 638, *816*, 1101, 1144

<drop collation statement> • 104, 164, 557, **641**, 642, 785, *816*, 1101, 1148

<drop column default clause> • *597*, **599**, 1140

<drop column definition> • *594*, **604**, 605, 1135

<drop column scope clause> • *597*, **601**, 602, 1140, 1160

<drop data type statement> • 86, 104, 557, **696**, 698, 785, *816*, 1101, 1135, 1236

<drop domain constraint definition> • *628*, **632**, 1145, 1150

<drop domain default clause> • *628*, **630**, 1150

<drop domain statement> • 103, 557, **633**, 634, 732, 737, 784, *816*, 1101, 1138

<drop method specification> • *676*, **692**, 695

<drop role statement> • 104, 120, **768**, *816*, 1101, 1184

<drop routine statement> • ?, 86, 104, 557, 602, 608, 612, 624, 651, 697, **727**, 728, 732, 736, 745, 747, 748, 785, *816*, 1101, 1135, 1157, 1236

<drop schema statement> • 103, **556**, 558, *815*, 1101, 1140

<drop sequence generator statement> • 104, 557, **752**, *816*, 1101, 1180

<drop table constraint definition> • *594*, **607**, 609, 1140

<drop table statement> • 103, 556, **610**, 611, 612, 732, 784, *815*, 881, 1101, 1135, 1231

<drop transform element list> • *741*, **744**, 745

<drop transform statement> • 104, 697, 728, **746**, 748, *816*, 1101, 1167

<drop transliteration statement> • 104, 164, 557, **646**, 647, *816*, 1101, 1149

<drop trigger statement> • 104, 557, 605, 609, 612, 624, 651, **657**, 732, 737, 784, *816*, 1101, 1180

<drop user-defined cast statement> • 104, 697, 698, 728, **731**, 732, *816*, 1101, 1164
<drop user-defined ordering statement> • 104, 728, **736**, 737, *816*, 1101, 1168
<drop view statement> • 103, 556, 602, 608, 612, **622**, 623, 624, 651, 732, 737, 783, *815*, 1101, 1135, 1231
DYNAMIC • *145*, *700*, 704, *789*, 791, 793, *1030*, 1133
<dynamic close statement> • 99, 105, 109, 112, 114, 115, *817*, **1018**, 1101, 1131
<dynamic cursor name> • **161**, 167, 168, 169, *1014*, *1016*, *1018*, *1019*, *1021*, 1022, 1130, 1206
<dynamic declare cursor> • 10, 84, 97, 98, 100, 105, 110, 112, 114, 115, 163, *790*, 791, 833, 837, 838, 986, **1009**, 1010, *1029*, 1033, 1035, 1039, 1131
<dynamic delete statement: positioned> • 100, 101, 105, 109, 112, 114, 115, 590, *817*, 856, 866, 873, 893, 894, **1019**, 1020, 1101, 1131
<dynamic fetch statement> • ?, ?, ?, ?, ?, ?, 84, 92, 100, 105, 109, 112, 114, 115, *817*, 976, 1001, 1002, **1016**, 1101, 1131, 1206
<dynamic open statement> • 84, 99, 105, 109, 112, 114, 115, *817*, 837, 838, 976, 997, **1014**, 1015, 1101, 1131, 1206
<dynamic parameter specification> • 83, 88, *187*, **188**, 189, 190, 191, 291, 366, 456, 505, 508, 526, 560, 570, 592, 613, 648, 655, 707, *882*, 885, 979, 985, 986, 991, 992, 993, 994, 995, 997, 998, 1002, 1007, 1008, 1014, 1089, 1130, 1132, 1206, 1214
<dynamic select statement> • ?, 84, 105, 110, 113, 114, 115, 818, *977*, **978**, 992, 993, 997, 1001, 1002, 1006, 1008, 1101, 1106
<dynamic single row select statement> • ?, 84, 105, 109, 113, 114, 115, 227, 818, *977*, 978, 992, 993, 1001, 1002, 1006, 1007, 1008, **1017**, 1101, 1106, 1131
dynamic SQL error • ?, 968, 971, 972, 974, 975, 976, 986, 997, 998, 999, 1000, 1002, 1003, 1004, 1006, 1007, 1012, 1014, 1117
<dynamic update statement: positioned> • 100, 101, 106, 109, 112, 114, 116, 590, *817*, 856, 866, 873, 893, 895, **1021**, 1022, 1102, 1131, 1132
DYNAMIC_FUNCTION • *144*, 966, *970*, 973, 992, *1095*, 1100, 1109
DYNAMIC_FUNCTION_CODE • *144*, 966, *970*, 973, 992, *1095*, 1100

— E —

E • *151*
EACH • *145*, *653*, 655, 656, 1180, 1252
ELEMENT • *145*, *246*
ELSE • *145*, 205, *208*, 209, 210, 304, 305, 307, 387, 1110
<else clause> • **208**, 210, 211
<embedded authorization clause> • **1030**, 1034
<embedded authorization declaration> • 82, *1029*, **1030**, 1032, 1034, 1039, 1040, 1133, 1206, 1215
<embedded authorization identifier> • **1030**
<embedded character set declaration> • **1030**, 1032, 1034, 1039, 1040, 1145, 1206
<embedded collation specification> • *1029*, **1030**, 1032, 1034
<embedded exception declaration> • 108, *1029*, 1035, 1039, **1041**, 1042, 1043, 1044, 1145, 1234
<embedded path specification> • *1029*, **1030**, 1032, 1034, 1040, 1161, 1206
<embedded SQL Ada program> • *1029*, 1031, 1033, 1034, 1039, 1042, **1045**, 1046, 1047, 1049, 1050, 1129, 1177
<embedded SQL begin declare> • **1030**, 1031, 1032, 1039, 1078
<embedded SQL C program> • *1029*, 1031, 1033, 1034, 1039, 1042, **1052**, 1054, 1055, 1057, 1059, 1129, 1206
<embedded SQL COBOL program> • *1029*, 1031, 1033, 1034, 1039, 1042, **1060**, 1061, 1062, 1064, 1065, 1129, 1207
<embedded SQL declare section> • 82, **1030**, 1031, 1032, 1039, 1046, 1054, 1061, 1067, 1073, 1078, 1084
<embedded SQL end declare> • **1030**, 1031, 1032, 1039, 1078
<embedded SQL Fortran program> • *1029*, 1031, 1033, 1034, 1039, 1042, **1066**, 1067, 1070, 1129, 1173
<embedded SQL host program> • ?, 82, 83, **1029**, 1031, 1032, 1033, 1040, 1042, 1043, 1206, 1215, 1234, 1251
<embedded SQL MUMPS declare> • **1030**, 1031, 1039
<embedded SQL MUMPS program> • *1029*, 1031, 1033, 1034, 1042, **1072**, 1073, 1075, 1076, 1129, 1174
<embedded SQL Pascal program> • *1029*, 1031, 1033, 1034, 1039, 1042, **1077**, 1078, 1080, 1081, 1130, 1174
<embedded SQL PL/I program> • *1029*, 1031, 1033, 1034, 1039, 1042, 1043, **1083**, 1084, 1087, 1088, 1130, 1207
<embedded SQL statement> • 82, 83, 189, **1029**, 1031, 1032, 1033, 1039, 1046, 1054, 1061, 1062, 1067, 1073, 1078, 1084
<embedded transform group specification> • *1029*, **1030**, 1032, 1034, 1035, 1037, 1040, 1167
<embedded variable name> • *187*, *188*, 189, 190, 191, 655, 707, *882*, **1031**, 1034, 1035, 1039, 1215, 1234
<embedded variable specification> • *187*, **188**, 190, 191, 508, 523, 560, 570, 592, 613, 648, 844, 846, 849, 851, 885, 997, 1001, 1089
<empty grouping set> • **340**, 341, 342, 343, 344, 345, 346, 348, 1185
<empty specification> • **193**, 194, 212, 215, 313, 575, 576, 577, 859, 864, 877, 1161, 1168
END • *145*, 205, *208*, 209, 304, 305, 307, 387, *653*, *1030*, 1038, 1047, 1079, 1110
<end field> • 176, 293, 454, **497**, 498, 499, 500
END-EXEC • *145*

EQUALS • 17, 41, 44, *144*, 671, *733*, 734, 735, 736
<equals operator> • 7, 23, 29, 71, *138*, **139**, *395*, 399, 401, 422, 466, 471, 473, *875*, *970*, *973*, *1045*, *1054*, *1095*, *1096*, 1232
<equals ordering form> • **733**, 734
error in assignment • 971, 974, 1115
ESCAPE • *145*, *270*, 279, 280, *405*, 406, 407, 408, *411*, 412, 413, 414, 979
<escape character> • *270*, 273, 279, **405**, 407, 410, *411*, 412, 413, 467, 984, 1139, 1225
escape character conflict • 413, 1115
<escape octet> • **405**, 406, 408, 409, 467, 984
<escaped character> • *411*, **412**, 413
EVERY • 64, *145*, *535*, 537, 540, 544, 574, 580, 581, 585, 1172
<exact numeric literal> • 30, **151**, 155, 156, 218, 220, 576, 1195, 1222
<exact numeric type> • 30, 31, **172**, 175, 179, 180, 805, 995, 1050, 1196, 1221, 1222
EXCEPT • 23, 29, 50, 77, *145*, 248, 249, *304*, 305, 306, *371*, 374, 375, 377, 378, 379, 380, 381, 382, 383, 384, 400, 469, 471, 905, 912, 1139, 1170, 1186, 1231, 1235
EXCLUDE • *144*, *352*, 359, 360
EXCLUDING • *144*, *560*, 561
<exclusive user-defined type specification> • **439**
EXEC • ?, *145*, *1029*, *1030*, 1031
EXECUTE • ?, ?, ?, ?, ?, ?, 118, 119, *145*, 215, 237, 505, 619, 620, 644, 716, 717, 728, 753, 755, 756, 757, *761*, 762, 775, 776, 777, 779, 780, *1006*, *1008*, 1039
<execute immediate statement> • 11, 81, 84, 93, 107, 110, 111, 128, 129, 162, 163, 164, 165, 475, 506, 507, *817*, 948, 949, 951, 952, **1008**, 1102, 1131, 1194, 1195, 1214, 1215
<execute statement> • ?, 11, 84, 108, 110, 111, *817*, 952, 976, 997, 1001, **1006**, 1007, 1102, 1131, 1132, 1206
<existing collation name> • **639**, 780, 1202
<existing transliteration name> • 281, **643**, 644, 1203
<existing window name> • **351**, 353, 354, 355, 360, 1189
EXISTS • *145*, *423*, 582, 905, 912, 1186
<exists predicate> • 23, 29, 362, *393*, 394, **423**, 604, 1186, 1225
EXP • *145*, *255*, 260
<explicit row value constructor> • **311**, *312*, 313, *314*, 315, 1147, 1176
<explicit table> • 54, **371**, 375, 376, 378, 384, 550, 1148, 1212
<exponent> • 30, **151**, 157
<exponential function> • 32, *253*, **255**, 257, 260, 263, 1189, 1198
<extended cursor name> • ?, **161**, 166, 168, 169, *991*, 992, *1011*, 1012, 1013, 1025, 1132
<extended statement name> • **161**, 166, 167, 168, 169, 838, 986, *1011*, 1012, 1132
EXTERNAL • *145*, *701*, 709, 720, 721, 1202
<external body reference> • 87, **701**, 704
external routine exception • 513, 518, 1108, 1118
external routine invocation exception • 517, 1108, 1118
<external routine name> • 87, **160**, 167, *701*, 709, 718, 719, 720, 722, *724*, 725, 726, 1190
<external security clause> • **701**, 709, 720, 721, 1183
<externally-invoked procedure> • ?, ?, ?, 11, 81, 82, 83, 93, 96, 111, 112, 121, 127, 135, 554, *790*, 791, 792, **795**, 796, 797, 798, 807, 808, 818, 819, 820, 833, 935, 936, 939, 940, 941, 1034, 1035, 1039, 1091, 1193, 1194, 1205, 1215, 1246, 1251
EXTRACT • *145*, *254*, 258
<extract expression> • 31, 39, *253*, **254**, 256, 258, 263, 1136, 1142, 1198
<extract field> • **254**, 256, 258
<extract source> • **254**, 256, 257, 258

— F —

Feature F032, “CASCADE drop behavior” • 612, 624, 698, 728, 1135
Feature F033, “ALTER TABLE statement: DROP COLUMN clause” • 605, 1135
Feature F034, “Extended REVOKE statement” • 785, 1135
Feature F052, “Intervals and datetime arithmetic” • 158, 181, 263, 287, 294, 295, 500, 1136
Feature F053, “OVERLAPS predicate” • 431, 1136
Feature F111, “Isolation levels other than SERIALIZABLE” • 926, 1137
Feature F121, “Basic diagnostics management” • 926, 1111, 1137
Feature F122, “Enhanced diagnostics management” • 1111, 1137
Feature F123, “All diagnostics” • 1111, 1137
Feature F131, “Grouped operations” • ?, ?, 1222, 1224, 1228
Feature F171, “Multiple schemas per user” • 555, 1137
Feature F191, “Referential delete actions” • 590, 1137
Feature F222, “INSERT statement: DEFAULT VALUES clause” • 861, 1137
Feature F251, “Domain support” • 169, 191, 626, 634, 1137, 1138
Feature F262, “Extended CASE expression” • 211, 1138
Feature F263, “Comma-separated predicates in simple CASE expression” • 211, 1138
Feature F271, “Compound character literals” • 158, 1138
Feature F281, “LIKE enhancements” • 409, 410, 1138, 1139, 1171
Feature F291, “UNIQUE predicate” • 424, 1139

Feature F301, “CORRESPONDING in query expressions” • 384, 1139
Feature F302, “INTERSECT table operator” • 384, 1139
Feature F304, “EXCEPT ALL table operator” • 384, 1139
Feature F312, “MERGE statement” • 868, 1139
Feature F321, “User authorization” • 191, 578, 945, 1139, 1140
Feature F361, “Subprogram support” • 1040, 1140
Feature F381, “Extended schema manipulation” • 558, 597, 598, 599, 600, 602, 606, 609, 726, 1140, 1141
Feature F391, “Long identifiers” • 149, 1141
Feature F392, “Unicode escapes in identifiers” • 149, 1141
Feature F393, “Unicode escapes in literals” • 158, 1141
Feature F394, “Optional normal form specification” • 284, 426, 1141
Feature F401, “Extended joined table” • 337, 338, 1141
Feature F402, “Named column joins for LOBs, arrays, and multisets” • 338, 1141, 1142
Feature F403, “Partitioned join tables” • 338, 1142
Feature F411, “Time zone specification” • 158, 181, 263, 287, 289, 947, 1142
Feature F421, “National character” • 158, 181, 226, 263, 410, 1142, 1143, 1175
Feature F431, “Read-only scrollable cursors” • 836, 846, 1143
Feature F441, “Extended set function support” • 339, 544, 545, 1143, 1144
Feature F442, “Mixed column references in set functions” • 545, 1144
Feature F451, “Character set definition” • 636, 638, 1144
Feature F461, “Named character sets” • 169, 528, 555, 794, 951, 1040, 1144, 1145
Feature F491, “Constraint management” • 169, 534, 632, 1044, 1145
Feature F521, “Assertions” • 649, 652, 1145
Feature F531, “Temporary tables” • 568, 881, 1146
Feature F555, “Enhanced seconds precision” • 158, 181, 289, 1146
Feature F561, “Full value expressions” • 404, 544, 1146
Feature F571, “Truth value tests” • 299, 1146, 1147
Feature F591, “Derived tables” • 329, 1147
Feature F611, “Indicator data types” • 191, 1147, 1197
Feature F641, “Row and table constructors” • 313, 317, 1147
Feature F651, “Catalog name qualifiers” • 168, 948, 1147, 1148
Feature F661, “Simple tables” • 384, 1148
Feature F671, “Subqueries in CHECK constraints” • 593, 1148
Feature F672, “Retrospective check constraints” • 593, 649, 1148
Feature F690, “Collation support” • 168, 532, 640, 642, 1148
Feature F692, “Extended collation support” • 574, 627, 675, 1148, 1149
Feature F693, “SQL-session and client module collations” • 192, 792, 955, 1149
Feature F695, “Translation support” • 168, 284, 645, 647, 1149
Feature F701, “Referential update actions” • 591, 1150
Feature F711, “ALTER domain” • 628, 629, 630, 631, 632, 1150
Feature F721, “Deferrable constraints” • ?, 534, 928, 1150
Feature F731, “INSERT column privileges” • 763, 1150
Feature F741, “Referential MATCH types” • 429, 590, 1150, 1151
Feature F751, “View CHECK enhancements” • 621, 1151
Feature F761, “Session management” • 944, 948, 950, 951, 1151
Feature F762, “CURRENT_CATALOG” • 191, 578, 1151
Feature F763, “CURRENT_SCHEMA” • 191, 578, 1151, 1152
Feature F771, “Connection management” • 168, 937, 939, 941, 1152
Feature F781, “Self-referencing operations” • 856, 861, 868, 874, 879, 1152, 1153
Feature F791, “Insensitive cursors” • 836, 1153, 1181
Feature F801, “Full set function” • 369, 1153
Feature F821, “Local table references” • 169, 200, 1153
Feature F831, “Full cursor update” • 841, 870, 1153
Feature F841, “LIKE_REGEX predicate” • 418, 1153, 1154
Feature F842, “OCCURENCES_REGEX function” • 264, 1154
Feature F843, “POSITION_REGEX function” • 264, 1154
Feature F844, “SUBSTRING_REGEXP” • 284, 1154
Feature F845, “TRANSLATE_REGEX” • 284, 1154
Feature F846, “Octet support in regular expression operators” • 264, 284, 1154
Feature F847, “Nonconstant regular expression” • 264, 284, 418, 1154, 1155
<factor> • **251**, *290*, 291, 292
FALSE • *145*, *153*, 158, 219, 221, *296*, 298, 399, 810, 812
feature not supported • 923, 935, 938, 1118
FETCH • *145*, *843*, *1016*
<fetch orientation> • **843**, 844, 846, *1016*, 1143
<fetch statement> • 92, 100, 105, 109, 112, 114, 115, 791, *816*, 820, **843**, 846, 892, 894, 1102, 1143, 1213, 1228
<fetch target list> • **843**, 844, 845, 1213
<field definition> • *173*, 181, **183**, 184, 1175
<field name> • 45, 46, **161**, 167, 168, *183*, *229*, 312, 326, 331, 332, 333, 368, 454, 455, 568, 594, 876, 1175
<field reference> • *185*, 186, 197, **229**, 367, 655, 1176

FILTER • *145*, *535*
<filter clause> • 63, 202, **535**, 538, 539, 545, 1189
FINAL • *144*, *659*, 660, 661, 673, 1158
<finality> • **658**, 660, 661, 673, 1158
FIRST • ?, 60, 61, *144*, 356, 357, 358, *385*, *546*, 547, *843*, 889, 890
FLAG • *144*, *253*, *270*, *271*, *417*
FLOAT • 14, 15, 30, *145*, *172*, 176, 180, 458, 462, 824, 825, 827, 828, 829, 830, 958, 960, 975, *1084*, 1087, 1196, 1197
FLOOR • *145*, *255*
<floor function> • 32, *253*, **255**, 257, 260, 264, 1189, 1198
<fold> • 22, **270**, 272, 273, 277, 280, 281, 1223, 1245
FOLLOWING • *144*, 205, *352*, 353, 356, 357, 358, 359
FOR • ?, ?, ?, 85, 99, *145*, *188*, *227*, 258, *270*, *271*, *272*, 275, 277, 525, *529*, *639*, *643*, *653*, 655, 656, 670, 671, 697, *700*, *701*, 728, *733*, *736*, *738*, *741*, *746*, *769*, 773, 774, 783, 785, 786, *789*, 791, 792, 793, *833*, *837*, 839, 840, 841, 888, *953*, *954*, 979, 980, 987, *1009*, *1011*, *1030*, 1133, 1135, 1153, 1161, 1177, 1180, 1252
FOREIGN • *145*, 572, *583*, 1251, 1252
FORTRAN • *144*, 476, *501*, 517, 518, 715, 723, 725, 793, 808, 809, 810, 811, 812, 813, 1034, 1133, 1134
<Fortran array locator variable> • *1066*, **1067**, 1069, 1071, 1165
<Fortran BINARY variable> • **1066**, 1068, 1071, 1171
<Fortran BLOB locator variable> • *1066*, **1067**, 1069, 1071, 1174
<Fortran BLOB variable> • *1066*, **1067**, 1068, 1071, 1173
<Fortran CLOB locator variable> • *1066*, **1067**, 1069, 1071, 1174
<Fortran CLOB variable> • **1066**, 1067, 1068, 1071, 1173
<Fortran derived type specification> • **1066**
<Fortran host identifier> • *1031*, **1066**, 1067, 1068, 1070
<Fortran multiset locator variable> • *1066*, **1067**, 1069, 1071, 1166
<Fortran REF variable> • *1066*, **1067**, 1070, 1159
<Fortran type specification> • **1066**, 1067, 1068
<Fortran user-defined type locator variable> • *1066*, **1067**, 1069, 1071, 1164
<Fortran user-defined type variable> • *1066*, **1067**, 1068, 1071, 1167
<Fortran VARBINARY variable> • **1066**, 1068, 1071, 1171
<Fortran variable definition> • *1030*, **1066**, 1067, 1070
FOUND • *144*, *1041*, 1042, 1044
FREE • *145*, *882*
<free locator statement> • 95, 105, 108, 109, 112, 113, 116, 709, *816*, **882**, *977*, 985, 1102, 1187, 1203, 1246
FROM • 58, 73, 76, *145*, 216, 221, 222, 223, 224, 225, 240, 244, 246, *253*, *254*, 258, *270*, *271*, *272*, 275, 276, 277, 304, 305, 307, 308, *319*, 323, 324, 331, 332, 336, 337, 347, 365, 375, 377, 382, 386, 387, 388, *432*, 543, 544, 568, 574, 580, 581, 582, 585, 605, 611, 614, 621, 634, 638, *639*, 642, *643*, 646, *659*, 671, 698, *700*, 728, *738*, *744*, 752, 768, *769*, *843*, *852*, *854*, 867, 902, 905, 912, 931, 935, 936, 945, 946, 948, 949, 951, 952, 953, 954, 967, *977*, 979, 980, 986, 1008, 1011, *1016*, *1019*, *1023*, 1104, 1105, 1252
<from clause> • v, 58, 75, 199, 206, *318*, **319**, 320, 325, 339, 341, 347, 354, 364, 367, 368, 373, 374, 386, 548, 616, 898, 903, 909, 1233, 1234
<from constructor> • 227, **857**, 859
<from default> • **857**, 861, 1137
<from sql> • **738**, 739, 740, 742, 743, 744
<from sql function> • **738**, 739, 742, 743
<from subquery> • 227, **857**, 858, 859, 861, 863, 1152
FULL • 17, 41, 44, 67, 68, 72, 73, *145*, *331*, 333, 334, 336, 338, 374, *427*, 428, 473, *583*, 585, 586, 588, 671, *733*, 734, 735, 736, 1141, 1168
<full ordering form> • **733**
FUNCTION • 42, *145*, *529*, 530, 531, 670, 671, 672, *699*, 735
function executed no return statement • 515, 1120
<function specification> • 87, **699**, 704, 708, 1236
FUSION • *145*, *535*, 537, 540, 545, 1170

— G —

G • *141*, *144*, 174
GENERAL • 88, *144*, 517, 520, 521, 672, *701*, 714, 718, 720, 725, 726, 1156
<general literal> • **150**
<general set function> • 23, 29, 202, 471, 472, **535**, 536, 537, 539, 544, 545, 1143, 1144, 1146
<general value specification> • 69, 128, 129, **187**, 190, 191, 574, 810, 811, *997*, 1130, 1138, 1139, 1151, 1152, 1160, 1197
<generalized expression> • **504**, 507, 508, 509, 511, 525, 1155
<generalized invocation> • **232**, 233
GENERATED • 59, *144*, *560*, 561, 562, 563, 568, *570*, 615, 616, 619, *659*, 1159
<generation clause> • **570**, 571, 573, 574, 1179
<generation expression> • 562, **570**, 571
<generation option> • **560**, 561
<generation rule> • **570**
GET • *145*, *635*, *970*, *1095*
<get descriptor information> • **970**, 972
<get descriptor statement> • ?, 84, 107, 109, *817*, **970**, 971, 972, 1102, 1130, 1214
<get diagnostics statement> • 96, 107, *817*, **1095**, 1097, 1099, 1106, 1111, 1122, 1137, 1215, 1219
<get header information> • **970**, 971, 972

<get item information> • **970**, 971, 972

GLOBAL • ?, 54, 98, *145*, *162*, *559*, 566, 1012, 1013, 1025

<global or local> • **559**

GO • *144*, *1041*, 1042, 1043, 1044

<go to> • **1041**, 1042, 1043, 1044

GOTO • *144*, *1041*

<goto target> • **1041**, 1042

GRANT • ?, ?, ?, ?, ?, 119, *145*, 527, 568, 621, 635, 672, 754, 755, 756, 757, *758*, 759, *766*, *769*, 772, 773, 774, 783, 785, 1135

<grant privilege statement> • 103, 119, *753*, **758**, 760, 1102, 1157, 1161, 1227

<grant role statement> • 104, 120, *753*, **766**, 767, 1102, 1184

<grant statement> • 527, *554*, 568, 621, 635, **753**, 754, 755, 756, 759, 762, *815*, 1235

GRANTED • *144*, *758*, 759, *766*, *769*, 770

<grantee> • *758*, 759, **761**, 762, *766*, 767, *769*, 770, 785, 786, 1136

<grantor> • *758*, **761**, 762, 763, *765*, *766*, *769*, 770, 787, 1184

<greater than operator> • *138*, **139**, *395*, 396, 547

<greater than or equals operator> • *142*, **143**, *395*, 396

GROUP • *145*, *253*, *270*, *340*, 341, 342, 345, 346, 347, 348, 349, *352*, 360, 366, *536*, *701*, *953*, 1185

<group by clause> • v, ?, 23, 29, 54, 75, 76, 206, *318*, 325, **340**, 341, 342, 344, 347, 348, 349, 354, 364, 366, 368, 469, 607, 616, 1185, 1211, 1224, 1233, 1234

<group name> • *701*, 706, 710, 711, 713, 714, 715, 728, **738**, 739, 740, *741*, 742, 744, *746*, 790, 791, 953, 1032, 1035, 1037

<group specification> • **701**, 706, 710, 711, 713, 714, 715, 790, 791, 1032, 1035, 1037

GROUPING • 62, 76, *145*, *202*, 203, *340*, 342, 343, 344, 345, 346, 348, 1126, 1127, 1185

<grouping column reference> • **340**, 341, 342, 344, 346

<grouping column reference list> • 76, **340**, 341, 346, 348, 650, 1185

<grouping element> • **340**, 344, 348, 1185

<grouping element list> • **340**, 345

<grouping operation> • **202**, 203, 342, 346, 1185

<grouping set> • **340**, 342, 343, 344, 345

<grouping set list> • **340**, 348, 1185

<grouping sets specification> • **340**, 341, 342, 343, 344, 345, 348, 1185

— H —

HAVING • *145*, *349*

<having clause> • v, 62, 76, 200, 202, 206, *318*, 325, 331, 339, 341, **349**, 354, 364, 366, 368, 374, 616, 1225, 1233, 1234

<header item name> • **970**, 971, 972, *973*, 976

held cursor requires same isolation level • 923, 1119

<hexit> • *142*, 147, **151**, 153, 154, 155, 158, 1172

HIERARCHY • 118, *144*, 243, 327, 328, 568, 620, 621, 754, 755, 757, *758*, 759, 760, *769*, 773, 774, 775, 776, 778, 779, 782, 783, 786, 855, 864, 865, 871, 872, 1161

<high value> • **411**, 413, 414

HOLD • 92, 98, 100, *145*, *835*, 836, 838, *883*, 888, 1013, 1187

<hold locator statement> • 95, 105, 108, 109, 112, 113, 116, 709, *816*, **883**, *977*, 985, 1102, 1187, 1203

<host identifier> • **1031**, 1033, 1039

<host label identifier> • **1041**, 1042, 1043, 1044

<host parameter data type> • 790, **795**, 797, 1034, 1036, 1164, 1165, 1166

<host parameter declaration> • 93, 460, 790, 791, **795**, 796, 798, 806, 807, 818, 833, 1034, 1035, 1036, 1039, 1215

<host parameter declaration list> • 790, **795**

<host parameter name> • **160**, 167, *187*, *188*, 190, 274, 460, 648, 655, 707, *795*, 796, 818, 833, 846, *882*, 883, 1034, 1036, 1215

<host parameter specification> • *187*, **188**, 189, 190, 274, 460, 508, 523, 560, 570, 592, 613, 844, 849, 851, 885, 997, 1001

<host PL/I label variable> • **1041**, 1042, 1043, 1044

<host variable definition> • **1030**, 1032, 1033, 1039, 1040, 1140

HOUR • 35, 128, *145*, 156, 178, 258, 286, 454, *497*, 498, 947, 976, 984, 985

<hours value> • **152**, 156

<hypothetical set function> • 204, **536**, 538, 542, 545, 1189, 1202

<hypothetical set function value expression list> • **536**, 538, 542, 543, 545, 1144

HZ • 1120

— I —

<identifier> • ?, ?, 45, 79, 84, 85, 126, **159**, *160*, *161*, *162*, 166, 186, *195*, 198, 322, 323, 325, 326, *361*, 362, 363, 364, 365, 370, *412*, 462, *653*, *659*, 661, 662, *738*, 936, 967, 986, 1011, 1097, 1099, 1108, 1109, 1183, 1195

<identifier body> • **141**, 148

<identifier chain> • 88, **195**, 197, 1183

<identifier extend> • **141**, 146

<identifier part> • **141**, 147, 149, 1141, 1224

<identifier start> • **141**, 146, 149, 1141

IDENTITY • *145*, *560*, 561, 562, *570*

<identity column specification> • 566, **570**, 572, 573, 574, 595, 1179

<identity option> • **560**, 561

IMMEDIATE • ?, ?, *144*, *533*, 534, 579, 625, 648, *927*, 931, *1008*, 1150, 1251, 1252
IMPLEMENTATION • *144*, 514, *701*, 709, 721, 1201
<implementation-defined character set name> • **527**, 528
implementation-defined classes • 1113
implementation-defined exception code • 1194
implementation-defined subclasses • 1113
<implicitly typed value specification> • **193**, *212*, *575*
IN • *145*, *253*, *254*, *270*, *271*, 387, *403*, 663, 682, 688, *699*, 707, 712, 713, 714, 979, 986, 995
<in predicate> • 248, 367, *393*, 394, **403**, 404, 466, 983, 1146, 1237
<in predicate part 2> • *208*, **403**
<in predicate value> • **403**
<in value list> • **403**, 404, 1146, 1237
<in-line window specification> • 60, **204**, 206, 364, 1209
inappropriate access mode for branch transaction • 924, 1119
inappropriate isolation level for branch transaction • 924, 1119
INCLUDING • *144*, *560*, 561, 562
<inclusive user-defined type specification> • **439**
INCREMENT • *144*, 562, *749*
<independent variable expression> • 64, 65, 202, *535*, **536**, 538, 541, 542, 544, 545, 1143, 1144
INDICATOR • *145*, *188*, *970*, 972, 974, 993, 998, 1004, 1214
indicator overflow • 445, 1115
<indicator parameter> • **188**, 190, 191, 366, 1147, 1197
<indicator variable> • **188**, 189, 191, 366, 1147, 1197
INITIALLY • *144*, *533*, 534, 579, 625, 648, 1150
INNER • 72, 73, *145*, *330*, 331, 333, 336
INOUT • *145*, *699*, 707, 712, 714, 986
INPUT • 87, *144*, 661, 664, 682, *701*, 704, *991*
<input using clause> • **997**, 1000, *1006*, *1014*, 1015, 1131, 1214
INSENSITIVE • 92, 98, 100, *145*, *835*, 836, 839, 886, 1153
INSERT • 118, 119, 120, 133, 134, 135, *145*, 548, 549, 550, 552, 568, 596, 605, *653*, 654, 759, *761*, 762, 763, 769, 770, 777, 781, *857*, 859, 861, *862*, 864, 881, 902, 916, 1137, 1150, 1200, 1252
<insert column list> • 777, 781, **857**, 858, 859, 860, 861, *862*, 863, 864, 867, 868, 982, 983, 1157
<insert columns and source> • **857**, 858, 859, 982
<insert statement> • 51, 59, 105, 109, 112, 113, 115, 131, 384, 568, 777, 778, 780, 781, 782, *816*, **857**, 858, 859, 861, *977*, 982, *1089*, 1102, 1104, 1105, 1148, 1152, 1177, 1215, 1227
<insertion target> • **857**, 861, 1177
INSTANCE • ?, ?, *144*, *529*, 531, *660*, 663, 673, 681, *692*, *700*, 1156
INSTANTIABLE • *144*, *658*, 660, 661, 668, 671, 672, 1156
<instantiable clause> • **658**, 660, 661, 672, 673, 1156, 1158
insufficient item descriptor areas • 993, 1121
INT • *145*, *172*, 174, 179, 799, 805, 807, 809, 812, 1047, 1048, 1049, *1072*, 1073, 1074, 1075, 1196
INTEGER • 14, 15, 30, 59, *145*, *172*, 174, 176, 180, 401, 458, 462, 512, 710, 713, 714, 734, 807, 824, 825, 827, 828, 829, 830, 958, 960, 985, 1049, 1058, 1064, *1066*, 1068, 1069, 1070, 1073, *1077*, 1079, 1080, 1081, 1087, 1196, 1251, 1252
integrity constraint violation • ?, 931, 1106, 1120
integrity constraint violation • 226, 587, 588, 589, 590, 820, 1042, 1106, 1109, 1118
INTERSECT • 23, 29, 50, 77, *145*, 248, 249, *304*, *371*, 374, 375, 376, 377, 379, 380, 383, 384, 400, 469, 471, 1139, 1186, 1212
INTERSECTION • *145*, 249, 306, 471, 472, *535*, 538, 540, 545, 1170
INTERVAL • 14, 15, 34, 35, 96, 128, *145*, *152*, 156, *173*, 181, 286, 287, 290, 291, 292, 293, 401, 430, 431, 459, 462, 824, 826, 827, 828, 830, 831, 947, 975, 976, 979, 980, 981, 984, 985
<interval absolute value function> • 39, **295**
<interval factor> • **290**, 291
interval field overflow • 225, 294, 447, 451, 1109, 1115
<interval fractional seconds precision> • 38, 157, 179, 257, **497**, 498, 499, 500, 958, 981, 995, 1200
<interval leading field precision> • 37, 38, 179, 290, 291, 292, 293, 401, *497*, **498**, 499, 500, 958, 980, 981, 995, 1199, 1200
<interval literal> • *150*, **152**, 156, 157, 158, 576, 1136
<interval primary> • *285*, 286, 287, **290**, 291, 1199
<interval qualifier> • 34, 37, *152*, 156, 157, *173*, 176, 179, 181, 225, *290*, 291, 292, 293, 294, 485, **497**, 498, 499, 500, 576, 822, 958, 965, 979, 995, 1136, 1200
<interval string> • *142*, 146, **152**
<interval term> • *285*, 286, 287, **290**, 291
<interval term 1> • **290**, 291, 292, 1199
<interval term 2> • **290**, 292
<interval type> • 34, *171*, **173**, 176, 181, 958, 995, 1136
<interval value expression> • *247*, 248, 249, *254*, 256, 257, 258, *285*, 286, 287, **290**, 291, 292, 293, 294, *295*, *947*, 984, 1136, 1211
<interval value expression 1> • **290**, 291, 292, 1199
<interval value function> • *290*, 291, **295**, 1136
interval value out of range • 539, 1115
INTO • *145*, 206, 568, *843*, *848*, *857*, *862*, *1001*, 1252
<into argument> • **1001**, 1002, 1005
<into arguments> • **1001**, 1002, 1005
<into descriptor> • ?, ?, **1001**, 1002, 1004
<introducer> • **150**, 154, 155

invalid argument for natural logarithm • 259, 1115
invalid argument for power function • 260, 1115
invalid argument for width bucket function • 261, 1115
invalid authorization specification • ?, 797, 935, 936, 945, 1118
invalid catalog name • 948, 1118
invalid character set name • 951, 1118
invalid character value for cast • 217, 218, 219, 220, 221, 226, 1109, 1115
invalid collation name • 954, 1118
invalid condition number • ?, 921, 924, 1106, 1118
invalid connection name • 936, 1118
invalid cursor name • 978, 992, 1011, 1014, 1016, 1018, 1019, 1021, 1118
invalid cursor state • 844, 885, 889, 891, 892, 894, 989, 1107, 1118
invalid DATA target • 974, 1117
invalid datetime format • 222, 223, 224, 1115
invalid DATETIME_INTERVAL_CODE • 976, 1117
invalid descriptor count • 998, 1002, 1117
invalid descriptor index • 968, 971, 974, 1117
invalid escape character • 279, 407, 413, 1115
invalid escape octet • 408, 1115
invalid escape sequence • 407, 408, 1115
invalid grantor • 770, 787, 1118
invalid indicator parameter value • 449, 1115
invalid interval format • 225, 1115
invalid LEVEL value • 974, 975, 1117
invalid parameter value • 797, 1115
invalid preceding or following size in window function • 356, 357, 358, 359, 1116
invalid regular expression • 279, 413, 1116
invalid repeat argument in a sample clause • 328, 1116
invalid role specification • 766, 946, 1118
invalid sample size • 328, 1116
invalid schema name • 949, 1118
invalid schema name list specification • 952, 1118
invalid specification • 882, 883, 930, 934, 1119, 1120
invalid SQL descriptor name • 967, 969, 971, 974, 992, 997, 1001, 1118
invalid SQL statement identifier • 986, 1119
invalid SQL statement name • 989, 992, 1006, 1012, 1014, 1119
invalid SQL-invoked procedure reference • 1012, 1119
invalid target type specification • 231, 1119
invalid time zone displacement value • 287, 947, 1116
invalid transaction state • 818, 819, 820, 855, 859, 865, 872, 892, 894, 921, 923, 924, 940, 945, 946, 1091, 1092, 1119
invalid transaction termination • 931, 933, 1119
invalid transform group name specification • 953, 1119
invalid use of escape character • 279, 413, 1116
invalid XQuery option flag • 480, 1116
invalid XQuery regular expression • 480, 1116
invalid XQuery replacement string • 483, 1116
<inverse distribution function> • **536**, 538, 543, 545, 1144, 1189, 1202
<inverse distribution function argument> • **536**, 543, 545, 1144
<inverse distribution function type> • **536**
INVOKER • *144*, 177, 200, 214, 215, 241, 277, 327, 528, 532, 618, *701*, 716, 717, 718, 719, 721, 853, 854, 859, 864, 870, 871, 1019, 1021, 1024, 1028
IS • ?, ?, ?, ?, 52, *145*, 209, 267, 280, *296*, 297, 298, 299, 304, 305, 307, *419*, *425*, *432*, *438*, *439*, 440, *559*, 572, 581, *659*, *1045*, *1046*, 1047, 1048, 1049, *1052*, *1053*, *1054*, 1055, 1056, 1057, 1058, *1060*, *1061*, 1062, 1063, 1064, *1066*, *1067*, 1068, 1069, 1070, *1072*, *1073*, 1074, 1075, *1077*, *1078*, 1079, 1080, *1083*, *1084*, 1085, 1086, 1087
ISOLATION • ?, ?, *144*, 921, *925*
<isolation level> • ?, ?, ?, **925**, 926, 943, 1137
<item number> • **970**, 971, 972, *973*, 974

— J —

JOIN • 73, *145*, *330*, 333, 1104, 1105
<join column list> • *330*, **331**, 332
<join condition> • 72, 73, 325, **330**, 331, 333, 378, 1232
<join specification> • 72, 73, **330**, 331, 333
<join type> • 73, **330**, 331, 374, 1231
<joined table> • v, 23, 29, 72, 74, 196, *321*, *322*, 324, 325, 326, 327, 328, 329, **330**, 331, 332, 333, 337, 338, 363, 373, 466, 467, 863, 1142, 1231, 1232

— K —

K • *141*, *144*, 174
KEY • 52, 67, 71, *144*, 564, 572, *581*, 582, *583*, 584, 606, 1251, 1252
<key word> • 14, 18, *141*, **144**, 149, 162, 296, 961, 963, 971, 973, 1191, 1207
KEY_MEMBER • *144*, *970*, 972, 973, 994, 1182
KEY_TYPE • *144*, *970*, 973, 992, 994
KIND • *1066*

— L —

LANGUAGE • *145*, *501*, 661, 663, 664, 670, 671, 672, 682, 683, 704
<language clause> • 40, 85, 87, **501**, 502, *660*, 664, 669, 682, *700*, 704, 715, 718, 719, 722, 723, *724*, 725, 726, *789*, 790, 792, 793, 794, 795, 796, 994, 1034, 1133, 1134, 1135

<language name> • 42, **501**, 669, 670, 685, 690, 718, 719, 725, 726, 796
LARGE • 14, 15, 18, 28, 96, *145*, *171*, *172*, 174, 175, 179, 226, 263, 410, 416, 443, 457, 462, 571, 810, 811, 813, 822, 823, 824, 825, 826, 827, 828, 829, 830, 958, 959, 963, 964, 975, 979, 1005, 1058, 1143, 1174, 1175
<large object length> • *171*, **172**, 174, 175, *271*, 276, *1046*, 1047, *1053*, *1054*, 1055, 1056, *1060*, *1061*, 1062, *1066*, *1067*, 1068, *1072*, *1073*, 1074, *1077*, 1078, 1079, *1083*, *1084*, 1085
<large object length token> • **141**, 147, *172*, 174
LAST • ?, 60, 61, *144*, 356, 357, 358, 547, *843*, 889, 890
LAST • *546*
LATERAL • *145*, *321*, 324, 329, 1186
<lateral derived table> • 74, **321**, 325, 326, 327, 328, 329, 331, 334, 1186
LEADING • *145*, *271*, 282, 284, *1061*
LEFT • 72, 73, *145*, *330*, 331, 333, 334, 336, 374
<left brace> • 23, *138*, **139**, *411*, 413
<left bracket> • 23, *138*, **139**, *412*, 413, 415, *1052*, *1077*
<left bracket or trigraph> • **139**, *173*, *188*, *193*, *245*, *302*, *308*, *875*
<left bracket trigraph> • **139**, *143*
<left paren> • ?, 23, **138**, *171*, *172*, *173*, *185*, *188*, 197, *202*, *204*, *208*, *212*, *230*, *232*, 234, *243*, *246*, *253*, *254*, *255*, *270*, *271*, *272*, *288*, *290*, *295*, *296*, *302*, *307*, *308*, *311*, *321*, *322*, *330*, *340*, *351*, *361*, *371*, 376, *390*, *403*, *411*, 413, 415, *439*, *497*, *504*, *529*, *535*, *536*, *559*, *560*, *570*, *581*, *583*, *592*, *613*, 619, *648*, *653*, *658*, *659*, *699*, *700*, *729*, *731*, *738*, *741*, *742*, *744*, *761*, *795*, *852*, *857*, *862*, *875*, 979, 980, *1029*, 1031, *1045*, *1046*, *1053*, *1054*, *1060*, *1061*, *1066*, *1067*, *1072*, *1073*, *1077*, *1083*, *1084*, 1225, 1226
LENGTH • *144*, 806, 813, 958, 959, *970*, 974, 975, 976, 994, 995, 1047, 1062, 1068, 1074, 1079, 1085, 1206, 1214
<length> • 83, 155, *171*, **172**, 174, 175, 177, *271*, 822, *1045*, *1046*, 1047, 1049, *1052*, *1053*, 1055, 1056, 1058, *1060*, *1061*, 1062, *1066*, 1067, 1068, 1070, *1072*, *1073*, 1074, *1077*, 1078, 1079, 1080, 1081, *1083*, 1085, 1087
<length expression> • 22, 29, 31, *253*, **254**, 257, 263, 1143, 1198
<less than operator> • *138*, **139**, *395*, 399, 547
<less than or equals operator> • *142*, **143**, *395*, 396
LEVEL • ?, ?, 86, *144*, 386, *700*, 704, 718, 921, *925*, 957, 967, *970*, 974, 975, 993, 994, 995, 998, 1002
<level of isolation> • ?, 923, 924, **925**, 926, 943, 1137, 1205
<levels clause> • **613**, 614, 621, 1151
LIKE • ?, ?, 22, *145*, *405*, 406, 407, 408, 409, 410, 416, *560*, 568, 569, 1138, 1139, 1171, 1179
<like clause> • ?, *559*, **560**, 561, 566, 568, 569, 1179
<like option> • **560**
<like options> • **560**, 561, 569, 1179
<like predicate> • 22, 23, 29, *393*, 394, **405**, 406, 408, 410, 466, 467, 1143, 1175, 1199, 1225
LIKE_REGEX • 23, *145*, *417*, 1219
LIST • 262
<list of attributes> • **659**, 662, 669
<literal> • 72, 75, **150**, 166, 169, *187*, 190, 217, 218, 219, 221, 222, 223, 224, 225, 226, 346, 362, 543, 544, *575*, 576, 577, 876, 885, 886, 968, 973, 978, 1093, 1109, 1110, 1132, 1245
LN • *145*, *255*, 260
LOCAL • ?, 54, 56, 98, 125, 126, *145*, *162*, 166, 167, 168, *285*, 286, 293, *559*, 566, *613*, 615, 619, 838, *880*, 905, 912, *923*, 924, *947*, 1025, 1181
<local or schema qualified name> • **159**, 162, 165
<local or schema qualifier> • **160**, 162, 169, 561, 880, 1153
<local qualified name> • **160**
<local qualifier> • **160**, 162, 169, 1153, 1188
LOCALTIME • *145*, *288*
LOCALTIMESTAMP • *145*, *288*, 289, 298, 593, 649, 1148
LOCATOR • 94, *144*, *659*, 663, 665, 681, 684, *699*, *882*, *883*, 958, 959, 960, 964, 972, 974, 975, 998, 1004, 1005, 1034, 1036, *1046*, 1048, 1049, *1054*, 1056, 1057, 1058, *1061*, 1063, 1064, *1067*, 1069, *1073*, 1074, 1075, *1077*, *1078*, 1079, 1080, *1084*, 1086
locator exception • 882, 883, 1119
<locator indication> • 90, 91, 663, 664, 665, 666, 667, 672, 673, 682, 683, 685, 687, 688, **699**, *700*, *701*, 705, 706, 707, 710, 711, 712, 713, 714, 718, 719, 721, 722, 790, *795*, 797, 1164, 1165, 1166, 1173, 1187, 1236, 1246
<locator reference> • **882**, *883*, 985
<low value> • **411**, 413, 414
LOWER • 22, *145*, *270*, 280, 412, 415

— M —

M • *141*, *144*, 174, 476, *501*, 502, 517, 518, 715, 723, 725, 793, 808, 809, 812, 1034, 1039, 1133, 1134
<major category> • **1041**, 1042
<mantissa> • 30, **151**, 157, 218, 220
MAP • 7, 16, 41, 44, *144*, 396, 398, 466, 467, 473, 671, *733*, 734, 735, 1168, 1209
<map category> • **733**, 734
<map function specification> • 16, 467, 473, **733**, 734
MATCH • 67, 68, *145*, *427*, 429, *583*, 585, 590, 1150, 1151
<match predicate> • 248, *393*, 394, **427**, 428, 429, 466, 467, 983, 1150, 1151
<match predicate part 2> • *209*, **427**
<match type> • 67, 68, **583**, 586, 588
MATCHED • *144*, *862*
MAX • 64, *145*, 249, 400, 473, *535*, 537, 540, 544, 545, *967*, 1176, 1206, 1212

<maximum cardinality> • **173**, 177, 182, 1162

maximum number of stacked diagnostics areas exceeded • 1112, 1117

<maximum returned result sets> • *700*, **701**, 717, 725

MAXVALUE • *144*, 492, 494, 562, *749*, 1200

MEMBER • *145*, *434*

<member> • **658**

<member list> • 177, **658**, 661, 662, 671, 672, 1155

<member name> • **529**

<member name alternatives> • **529**

<member predicate> • 248, 367, *393*, 394, **434**, 435, 436, 466, 467, 471, 1169

<member predicate part 2> • *209*, **434**

MERGE • *145*, *862*, 868, 1139

<merge correlation name> • **862**, 863, 867, 1104

<merge insert specification> • 777, 781, **862**, 983

<merge insert value element> • **862**, 864

<merge insert value list> • 227, 325, **862**, 864, 983

<merge operation specification> • **862**

<merge statement> • 51, 59, 105, 109, 112, 113, 115, 131, 325, 777, 778, 779, 780, 781, 782, *816*, **862**, 863, 864, 867, 868, 875, 893, 895, *977*, 983, *1089*, 1102, 1104, 1105, 1139, 1152, 1157, 1177, 1213, 1215

<merge update specification> • **862**

<merge when clause> • **862**, 865

<merge when matched clause> • 59, **862**, 865, 1104, 1105

<merge when not matched clause> • 59, **862**, 867, 1104, 1105

MESSAGE_LENGTH • *144*, *1096*, 1109

MESSAGE_OCTET_LENGTH • *144*, *1096*, 1109

MESSAGE_TEXT • *144*, *1096*, 1108, 1109, 1207

METHOD • ?, ?, *145*, *529*, 530, 531, *660*, 661, *692*, *700*, 722, 1156, 1158

<method characteristic> • 86, **660**

<method characteristics> • *659*, **660**, 664, 672, 681, 1156

<method invocation> • 87, *185*, 186, 197, **232**, 233, *236*, 241, 366, 679, 697, 724, 727, 775, 776, 777, 779, 780, 1155

<method name> • 40, 41, 42, 87, **160**, 167, *232*, *234*, *241*, *529*, 530, 531, 619, 620, *660*, 662, 664, 666, 667, 669, 670, 681, 685, 688, 690, *692*, 694, *700*, 703, 704, 754, 755, *875*, 876, 877, 879, 1157

<method reference> • 238, **241**, 242, 366, 601, 610, 619, 620, 622, 679, 697, 724, 727, 753, 755, 775, 776, 777, 778, 779, 780, 781, 1159

<method selection> • **232**, 233, 504, 506, 507

<method specification> • 40, 41, 42, 86, **659**, 662, 672, 673, 1187

<method specification designator> • 86, *699*, **700**, 702, 703, 716, 717, 719, 721, 722, 1155, 1158

<method specification list> • 40, 41, *658*, **659**, 662, 669, 670, 673, 1155

MIN • 64, *145*, 249, 400, 473, *535*, 537, 540, 544, 545, 1176, 1212

<minus sign> • 23, **138**, *143*, 147, *151*, *152*, *251*, 252, *285*, 287, *290*, 291, *412*, 413, 414, 499, 500, 980, 1136

MINUTE • 35, 128, *145*, 156, 178, 258, 286, 454, *497*, 498, 947, 976, 984

<minutes value> • **152**, 156

MINVALUE • *144*, 492, 494, 562, *749*, 1200

MOD • *145*, *254*, 263, 1185

MODIFIES • *145*, *701*, 704, 716, 720, 726

modifying SQL-data not permitted • 513, 515, 518, 1118, 1120

MODULE • ?, 55, *145*, *160*, 162, 165, *199*, 200, *794*, 880, 881, 1107, 1153

<module authorization clause> • 82, 165, **789**, 790, 793, 1034, 1133, 1195, 1206

<module authorization identifier> • 81, 117, 554, 780, **789**, 790, 796, 797, 935, 987, *1030*, 1205, 1215

<module character set specification> • 82, 780, **794**, 1034, 1145, 1204

<module collation specification> • **789**, 791, 792, 1149

<module collations> • **789**, *1030*, 1034

<module contents> • 97, 162, 163, *789*, **790**

<module name clause> • *789*, **794**, 1034, 1145

<module path specification> • 89, 720, **789**, 790, 792, 1034, 1160, 1204, 1206

<module transform group specification> • **789**, 790, 792, 1034, 1167

<modulus expression> • 32, *253*, **254**, 257, 259, 263, 979, 1185

MONTH • 35, 37, *145*, 156, 176, 178, 355, 356, 454, *497*, 498, 976, 980

<months value> • **152**, 156, 499

MORE • *144*, 525, *1095*, 1099

most specific type mismatch • 522, 1116

<multiple column assignment> • **875**, 876, 879, 984, 1190

<multiple group specification> • **701**, 706, 790, 1032

multiple server transactions • 923, 935, 938, 1118

<multiplier> • **141**, 147, 149, *172*, 174, 175, 1175

MULTISET • ?, ?, 13, 47, 48, 49, 50, 97, *145*, *173*, 181, *193*, 194, 216, 248, 249, *304*, 305, 306, 307, *308*, 459, 471, 485, 537, 575, 702, 824, 826, 827, 829, 830, 831, 859, 864, 877, 959, 960, 972, 974, 981, 998, 1004, 1005, 1168, 1170

<multiset element> • **308**, 309, 981

<multiset element list> • **308**, 309, 981

<multiset element reference> • 49, *185*, 186, **246**, 367, 1168

<multiset primary> • **304**, 305, 471

<multiset set function> • 50, 249, **307**, 471
<multiset term> • **304**, 305, 306, 471
<multiset type> • **173**, 177, 182, 673, 705, 797, *1046*, 1049, *1054*, 1057, *1061*, 1063, 1064, *1067*, 1069, *1073*, 1075, *1078*, 1080, *1084*, 1086, 1087, 1166, 1168, 1169, 1187
<multiset value constructor> • *185*, 186, **308**, 309, 1169, 1183
<multiset value constructor by enumeration> • **308**, 309
<multiset value constructor by query> • **308**, 309
<multiset value expression> • *246*, *247*, 248, 249, **304**, 305, 306, *307*, *434*, *436*, 471, 1169
<multiset value function> • *304*, 305, **307**, 1169
multiset value overflow • 539, 1116
MUMPS • *144*, *501*, 723, 793, 808, 1074, 1076, 1123, 1129, 1133, 1134
<MUMPS array locator variable> • *1072*, **1073**, 1075, 1076, 1165
<MUMPS BLOB locator variable> • *1072*, **1073**, 1074, 1076, 1174
<MUMPS BLOB variable> • *1072*, **1073**, 1074, 1076, 1174
<MUMPS character variable> • **1072**, 1073
<MUMPS CLOB locator variable> • *1072*, **1073**, 1075, 1076, 1174
<MUMPS CLOB variable> • **1072**, 1073, 1074, 1076, 1174
<MUMPS derived type specification> • **1072**, 1073
<MUMPS host identifier> • *1031*, **1072**, 1073, 1074, 1075
<MUMPS length specification> • **1072**, 1073
<MUMPS multiset locator variable> • *1072*, **1073**, 1075, 1076, 1166
<MUMPS numeric variable> • **1072**
<MUMPS REF variable> • *1072*, **1073**, 1075, 1076, 1159
<MUMPS type specification> • **1072**
<MUMPS user-defined type locator variable> • *1072*, **1073**, 1075, 1076, 1164
<MUMPS user-defined type variable> • *1072*, **1073**, 1074, 1076, 1168
<MUMPS VARBINARY variable> • **1072**, 1074, 1076, 1171
<MUMPS variable definition> • *1030*, **1072**, 1073, 1075
<mutated set clause> • **875**, 876, 877, 879, 984, 1157
<mutated target> • **875**, 876, 877

— N —

N • *150*
NAME • ?, ?, *144*, *701*, *724*, *970*, 973, 993, 994, 1214
<named columns join> • 72, 73, **330**, 331, 332, 333, 334, 336, 338, 1142
NAMES • *144*, *794*, *951*, *1030*
NATIONAL • 18, *145*, *172*, 174, 226, 263, 410, 805, 806, 1058, 1070, 1143, 1174, 1175, 1191, 1195
<national character large object type> • **172**, 182, 1172
<national character string literal> • *141*, 146, **150**, 153, 154, 158, 1142
<national character string type> • *171*, **172**, 181, 1143
NATURAL • 72, 73, *145*, *330*, 331, 333, 334, 336, 466, 467, 604
<natural join> • **330**, 331, 338, 374, 1141, 1142
<natural logarithm> • 32, *253*, **255**, 257, 259, 263, 1189, 1198
NCHAR • *145*, *172*, 174, 799, 805, 806, *1053*, 1055, 1058
NCLOB • *145*, *172*, 174, *1053*, 1055, 1058, *1060*, 1062
NESTING • *144*, *991*, 992, 993
<nesting option> • **991**
NEVER • 573
NEW • *145*, *236*, *653*, 654, 655, *700*, 704, 718
<new invocation> • 232, 233, **236**, 510
<new specification> • 87, *185*, 186, **236**, 237, 1155
<new transition table name> • 134, **653**, 654, 655, 656
<new transition variable name> • 134, 197, *653*, **654**, 655, 656
<new window name> • **351**, 352, 353, 354
<newline> • *143*, **144**, 146, 147, 154, 1195
NEXT • *144*, *227*, *843*, 889, 890, 1016
<next value expression> • 81, *185*, 186, **227**, 228, 248, 1179
NFC • *144*, *271*, 276, 425
NFD • *144*, *271*
NFKC • *144*, *271*
NFKD • *144*, *271*
NO • ?, ?, ?, 19, 25, 81, 92, 98, 100, *145*, *352*, 360, 400, 491, 492, 493, 494, 528, *560*, *583*, 585, *639*, 664, 672, 682, *701*, 704, 716, 726, *749*, *750*, 820, *835*, 888, *931*, *933*, *954*, 1013, 1156, 1200
no active SQL-transaction for branch transaction • 923, 1119
no additional result sets returned • 1012, 1119
no data • 93, 94, 477, 478, 820, 821, 844, 849, 856, 861, 868, 873, 890, 971, 1012, 1016, 1017, 1092, 1093, 1113, 1119
no subclass • 1113, 1114, 1115, 1117, 1118, 1119, 1120, 1121
<non-cycle mark value> • **385**, 386
<non-escaped character> • **411**, 413
<non-reserved word> • **144**, 1219
<non-second primary datetime field> • **497**, 499
noncharacter in UCS string • 177, 1116
<nondelimiter token> • 9, **141**, 147
<nondoublequote character> • **142**, 146
NONE • 41, *145*, 669, 733, 737, *946*
<nonparenthesized value expression primary> • **185**, 186, 248, *296*, 297, 298, 299, *314*, 1172

<nonquote character> • *143*, **150**, 154

<normal form> • 28, **271**, 276, 284, *425*, 426, 1141

NORMALIZE • *145*, 156, 267, *271*, 280

<normalize function> • 28, *270*, **271**, 272, 276, 278, 279, 284, 1176, 1192, 1198

<normalize function result length> • **271**, 276, 1198

NORMALIZED • *144*, 267, 280, *425*

<normalized predicate> • 28, *393*, **425**, 426, 1176, 1177, 1192

<normalized predicate part 2> • *209*, **425**

NOT • 22, 23, 32, 33, *145*, 209, *296*, 297, 298, 396, 397, *402*, *403*, *405*, 406, *411*, 412, *417*, *419*, *425*, 431, *432*, 433, *434*, *436*, *438*, *439*, 440, *533*, 534, 561, 562, 563, 564, *570*, 572, 579, 581, 582, 625, 648, *658*, *659*, 660, 664, 672, 673, 682, *701*, 704, 718, *862*, *1041*, 1042, 1044, 1150, 1156, 1158, 1179, 1188

<not equals operator> • 23, 29, *142*, **143**, *395*, 396, 401, 422, 466, 471, 473

NULL • ?, ?, 32, 52, 87, 132, *145*, *193*, *209*, 210, 215, 297, 304, 305, 307, 346, 347, 366, *419*, 449, 561, 562, 563, 564, *570*, 572, 581, 582, *583*, 585, 587, 588, 589, 661, 664, 675, 682, *701*, 704, *920*, 1188, 1225

<null ordering> • 61, 354, **546**, 547, 1188, 1202

<null predicate> • 23, *393*, 394, **419**, 420, 1236

<null predicate part 2> • *208*, **419**

null row not permitted in table • 316, 1116

<null specification> • **193**, 194, 313, 575, 920, 1228

null value eliminated in set function • 539, 541, 543, 1121

null value in array target • 523, 845, 850, 878, 1116

null value not allowed • 190, 517, 1116, 1118

null value substituted for mutator subject parameter • 510, 675, 1116

null value, no indicator parameter • 444, 972, 1116

<null-call clause> • 87, *660*, 664, 682, *700*, **701**, 704, *724*, 725, 726

NULLABLE • *144*, *970*, 973, 993, 994

NULLIF • *145*, *208*, 209

NULLS • 60, 61, *144*, 356, 357, 358, *546*, 547

NUMBER • *144*, *1095*, 1099

<number of conditions> • ?, ?, ?, ?, 921, 923, 924, **925**, 926, 944, 1214

NUMERIC • 14, 15, 30, *145*, *172*, 175, 176, 180, 458, 462, 806, 824, 825, 827, 828, 829, 830, 958, 959, 975, 979, 980, 985, 1064, 1196

<numeric primary> • **251**, 252, 1222

<numeric type> • *171*, **172**, 1221

<numeric value expression> • 64, *245*, *247*, 249, **251**, 252, *254*, *255*, 257, 259, 260, 261, *272*, *321*, *536*, 545, 979, 1144, 1198, 1222

<numeric value expression base> • **255**, 260

<numeric value expression dividend> • *254*, **255**, 259, 979

<numeric value expression divisor> • *254*, **255**, 257, 259, 979

<numeric value expression exponent> • **255**, 260

<numeric value function> • *251*, **253**, 257, 1245

numeric value out of range • 217, 252, 259, 260, 261, 262, 446, 451, 539, 540, 543, 1109, 1116

— O —

OBJECT • 14, 15, 18, 28, 96, *144*, *171*, *172*, 174, 175, 179, 226, 263, 410, 416, 443, 457, 462, 571, 810, 811, 813, 822, 823, 824, 825, 826, 827, 828, 829, 830, 958, 959, 963, 964, 975, 979, 1005, 1058, 1143, 1174, 1175

<object column> • 778, 781, 864, 866, 869, 870, 871, 872, 873, **875**, 876, 877, 878, 895, 1021, 1028

<object name> • 758, 759, **761**, 762, 763, 769, 785, 786, 1135, 1136, 1155, 1157, 1161

<object privileges> • 759, **761**, 762

OCCURRENCE • *253*, *270*, *271*

<occurrences> • **967**, 968, 971, 974, 992, 998, 1002, 1132, 1206

OCCURRENCES_REGEX • *146*, *253*, 1219

<octet length expression> • **254**, 259, 979, 1223, 1245

<octet like predicate> • **405**, 406, 408, 409, 984, 1175

<octet like predicate part 2> • *208*, **405**

<octet pattern> • **405**, 406, 467, 984

OCTET_LENGTH • *146*, *254*, 259, 277, *971*, 973, 994, 995, 1206, 1214

OCTETS • *144*, *172*, 262, 263, 278, 281, 1211

OF • ?, 99, *146*, *434*, *436*, *439*, 440, *559*, 566, 568, *613*, *653*, 808, *837*, *852*, *869*, *1019*, *1021*, *1023*, *1027*, *1077*, 1078, 1079, 1080, 1081, 1160

OLD • 86, *146*, *653*, 654, 655, *700*, 704

<old transition table name> • 134, **653**, 654, 655, 656

<old transition variable name> • 134, **653**, 654, 655, 656

ON • ?, ?, ?, ?, 56, 73, 87, *146*, *330*, 527, *559*, 565, 567, 568, *583*, 584, 585, 592, 605, 611, 621, 634, 635, 638, 642, 646, *653*, 661, 664, 682, 698, *701*, 704, 728, 752, 754, 755, 756, 759, *761*, *862*, *880*, 931, 1104, 1105, 1251, 1252

ONLY • ?, ?, ?, 17, 58, 74, 85, 99, *146*, 244, *321*, 328, 329, *439*, 616, 619, 671, *733*, 734, *789*, 791, 792, 793, *837*, 839, *852*, 853, 855, 857, 863, 864, 865, 866, 869, 871, 872, 873, 888, 893, 895, 899, 900, 910, 912, 924, *925*, 944, 987, 1019, 1021, 1023, 1027, *1030*, 1133, 1162, 1163

<only spec> • **321**, 326, 327, 329, 775, 776, 779, 782, 1162

OPEN • *146*, *842*, *1014*

<open statement> • 99, 105, 109, 112, 114, 115, 791, *816*, 833, **842**, 1034, 1035, 1039, 1102, 1215, 1228

OPTION • 56, 59, 118, 119, 120, *144*, 243, 327, 328, 568, *613*, 614, 615, 619, 620, 621, 672, 754, 755, 756, 757,

758, 759, 760, 763, *766*, *769*, 772, 773, 774, 775, 776, 778, 779, 782, 783, 785, 786, 855, 861, 864, 865, 866, 868, 871, 872, 873, 895, 905, 912, 1107, 1135, 1151, 1161, 1177
OPTIONS • *144*, *560*, *613*
OR • 33, 71, *146*, 210, *296*, 297, 298, 299, 304, 305, 396, 399, 402, 431
ORDER • ?, 99, *146*, *351*, 473, *536*, 543, 544, 671, *733*, 735, *837*, 839, 1168
<order by clause> • 23, 29, 62, 99, 195, 204, 205, 227, *302*, 325, 353, **837**, 839, 841, 869, 886, 1021, 1028, 1093, 1153, 1210, 1211, 1213, 1214
<ordered set function> • 202, 206, *535*, **536**
ORDERING • *144*, 671, 728, *733*, *736*
<ordering category> • **733**
<ordering form> • **733**
<ordering specification> • 61, 354, 356, 357, 358, 544, **546**
ORDINALITY • 74, *144*, *322*, 323, 324
<ordinary grouping set> • **340**, 342, 343, 344, 345, 346, 348, 1185
<ordinary grouping set list> • **340**, 342, 343
<original method specification> • 40, 42, **659**, 661, 664, 669, 672, *681*, 1156
OTHERS • *144*, *352*, 360
OUT • *146*, *699*, 707, 710, 711, 713, 715, 986
OUTER • *146*, *330*, 1104, 1105
<outer join type> • **330**, 331, 1231
OUTPUT • *144*, *991*
<output using clause> • ?, ?, **1001**, 1002, 1005, *1006*, *1016*, 1131, 1214
OVER • *146*, *204*, 205, 206, 543, 544
OVERLAPS • 39, *146*, *430*, 431, 1124, 1136
<overlaps predicate> • 39, 248, *393*, 394, **430**, 431, 473, 983, 1136
<overlaps predicate part 1> • *208*, 210, **430**
<overlaps predicate part 2> • *209*, 210, **430**
OVERLAY • 21, *146*, *271*, *272*, 284, 980, 1182
<override clause> • **857**, 858, 861, *862*, 863, 864, 1160
OVERRIDING • *144*, *659*, 663, *857*, 858, 860, 863, 867, 868
<overriding method specification> • 40, 42, **659**, 666, 670, *687*

— P —

P • *141*, *144*, 149, 175, 1126, 1175, 1246
PAD • 19, 25, *144*, 400, 528, *639*
<pad characteristic> • **639**, 640
PARAMETER • *146*, 366, 515, 517, 519, 520, 521, 664, 682, *700*, 706, 709, 714, 718, 720, 725, 726
<parameter mode> • 663, 682, 688, **699**, 707, 710, 711, 712, 713, 714, 715, 986, 995, 996, 1108, 1109
<parameter style> • 40, 42, 90, 664, 669, 670, 672, 685, 690, *700*, **701**, 706, 720, 1156
<parameter style clause> • *660*, 664, 682, **700**, *701*, 704, 705, *724*, 725, 726
<parameter type> • 459, 663, 683, 688, 689, **699**, 706, 707, 710, 711, 712, 713, 714, 715, 721, 722, 1163, 1164, 1165, 1166, 1173
<parameter using clause> • 997, **1006**, 1007
PARAMETER_MODE • *144*, 966, *971*, 973, 996, *1096*, 1108, 1109, 1206
PARAMETER_NAME • *144*, *1096*, 1108, 1109
PARAMETER_ORDINAL_POSITION • *144*, *971*, 973, 993, 996, *1096*, 1108, 1109, 1206
PARAMETER_SPECIFIC_CATALOG • *144*, *971*, 973, 993, 996, 1206
PARAMETER_SPECIFIC_NAME • *144*, *971*, 973, 993, 996, 1206
PARAMETER_SPECIFIC_SCHEMA • *144*, *971*, 973, 993, 996, 1206
<parenthesized boolean value expression> • **296**, 297, 298
<parenthesized joined table> • *321*, **322**
<parenthesized value expression> • **185**
PARTIAL • 67, 68, *144*, *427*, 428, *583*, 585, 586, 587, 589
<partial method specification> • *659*, **660**, 673, 681, 687, 1156
PARTITION • *146*, *330*, *351*
<partitioned join column reference> • **330**, 331
<partitioned join column reference list> • **330**
<partitioned join table> • **330**, 331, 334, 335, 338, 1142
PASCAL • *144*, 476, *501*, 517, 518, 715, 723, 725, 793, 808, 809, 811, 1034, 1081, 1134
<Pascal array locator variable> • *1077*, **1078**, 1080, 1081, 1165
<Pascal BINARY variable> • **1077**, 1079, 1082, 1171
<Pascal BLOB locator variable> • *1077*, **1078**, 1079, 1081, 1174
<Pascal BLOB variable> • **1077**, 1079, 1081, 1082, 1174
<Pascal CLOB locator variable> • **1077**, 1080, 1082, 1174
<Pascal CLOB variable> • **1077**, 1078, 1079, 1081, 1082, 1174
<Pascal derived type specification> • **1077**
<Pascal host identifier> • *1031*, **1077**, 1078, 1079, 1080
<Pascal multiset locator variable> • *1077*, **1078**, 1080, 1081, 1166
<Pascal REF variable> • *1077*, **1078**, 1080, 1081, 1159
<Pascal type specification> • **1077**, 1078
<Pascal user-defined type locator variable> • *1077*, **1078**, 1080, 1081, 1165

<Pascal user-defined type variable> • *1077*, **1078**, 1079, 1081, 1168

<Pascal variable definition> • *1030*, **1077**, 1078, 1080

PATH • *144*, *503*, *952*

<path column> • **385**, 386

<path specification> • **503**, *553*, 720, *789*, *1030*, 1160

<path-resolved user-defined type name> • 82, 92, 129, 163, *171*, **173**, 177, 181, 182, *187*, 189, 191, *230*, *234*, *236*, *439*, *504*, 507, 508, *559*, 563, 566, 567, 568, *613*, 615, 620, *658*, *701*, 706, 710, 711, 713, 714, 728, 790, 791, 885, 886, *953*, 1032, *1046*, 1048, 1051, *1053*, *1054*, 1057, 1059, *1061*, 1064, 1065, *1067*, 1069, 1071, *1073*, 1075, 1076, *1078*, 1080, 1081, *1084*, 1086, 1088, 1155, 1160, 1162, 1164, 1165, 1169

<percent> • 22, 23, *137*, **138**, 407, 408, 409, *411*, 413, 414

PERCENT_RANK • 62, 63, 65, *146*, *204*, 205, 207, 1188, 1193

PERCENTILE_CONT • 65, *146*, *536*, 539, 543, 544, 1202

PERCENTILE_DISC • 65, *146*, *536*, 544

<period> • 79, **138**, *151*, *153*, 155, *159*, *160*, *161*, 195, 197, *199*, 218, 220, *229*, *232*, *271*, *361*, 363, 499, 500, *504*, *660*, 805, *875*, *1060*

<PL/I array locator variable> • *1083*, **1084**, 1086, 1088, 1166

<PL/I BINARY variable> • **1083**, 1085, 1088, 1171

<PL/I BLOB locator variable> • *1083*, **1084**, 1086, 1088, 1174

<PL/I BLOB variable> • **1083**, 1085, 1088, 1174

<PL/I CLOB locator variable> • *1083*, **1084**, 1086, 1088, 1174

<PL/I CLOB variable> • **1083**, 1085, 1088, 1174

<PL/I derived type specification> • **1083**

<PL/I host identifier> • *1031*, **1083**, 1084, 1085, 1087

<PL/I multiset locator variable> • *1083*, **1084**, 1087, 1088, 1166

<PL/I REF variable> • *1083*, **1084**, 1087, 1088, 1159

<PL/I type fixed binary> • *1083*, **1084**

<PL/I type fixed decimal> • *1083*, **1084**

<PL/I type float binary> • *1083*, **1084**

<PL/I type specification> • **1083**, 1084, 1085, 1207

<PL/I user-defined type locator variable> • *1083*, **1084**, 1086, 1088, 1165

<PL/I user-defined type variable> • *1083*, **1084**, 1085, 1088, 1168

<PL/I VARBINARY variable> • **1083**, 1085, 1088, 1171

<PL/I variable definition> • *1030*, **1083**, 1084, 1085, 1087, 1207

PLACING • *144*, *271*, *272*, 980

PLI • *144*, 476, *501*, 517, 518, 715, 723, 725, 793, 808, 809, 810, 811, 812, 813, 1034, 1134, 1135

<plus sign> • 23, **138**, *142*, 147, *151*, *251*, 252, *285*, 287, *290*, *411*, 413, 414, 980, 1136

POSITION • 7, *146*, *254*, 979

<position expression> • 22, 23, 29, 31, **253**, 255, 466, 467, 979, 1198, 1224, 1245

POSITION_REGEX • 23, *146*, *253*, 1219

POWER • *146*, *255*, 257, 541, 542

<power function> • 32, *253*, **255**, 257, 260, 263, 1189, 1198

PRECEDING • *144*, 205, *352*, 353, 356, 357, 358, 359

PRECISION • 14, 15, 30, *146*, *172*, 176, 180, 458, 462, 824, 825, 827, 828, 829, 831, 958, 959, 960, *971*, 974, 975, 976, 995, 1049, 1058, *1066*, 1070, 1196, 1197, 1214

<precision> • 83, **172**, 174, 175, 176, 180, 806, 822, 979, *1072*, 1073, *1083*, 1087, 1195, 1196

<predefined type> • **171**, 462, *625*, *658*, *659*, 660, 661, *1046*, *1053*, *1061*, *1067*, *1073*, *1078*, *1084*, 1236

<predicate> • 33, *296*, 297, 298, **393**, 394, 407, 409, 417, 1226

<preparable dynamic cursor name> • 978, *1023*, **1025**, 1026, *1027*

<preparable dynamic delete statement: positioned> • 100, 101, 106, 109, 113, 115, 590, 856, 866, 873, 893, 894, *977*, **1023**, 1024, 1025, 1102, 1132

<preparable dynamic update statement: positioned> • 100, 101, 106, 109, 113, 116, 590, 856, 866, 873, 893, 895, *977*, 1025, **1027**, 1028, 1102, 1132

<preparable implementation-defined statement> • 113, *977*, **978**, 1206

<preparable SQL control statement> • **977**, 978

<preparable SQL data statement> • **977**

<preparable SQL schema statement> • **977**

<preparable SQL session statement> • **977**

<preparable SQL transaction statement> • **977**

<preparable statement> • 84, 89, 93, 128, 129, 131, 162, 163, 164, 165, 189, 475, 506, 507, 717, 837, 948, 949, 951, 952, **977**, 978, 1194, 1195

PREPARE • *146*, 519, *977*, 986, *989*, 1008

<prepare statement> • ?, 10, 11, 81, 84, 85, 93, 107, 109, 110, 128, 129, 162, 163, 164, 165, 167, 168, 189, 475, 506, 507, *817*, 837, 839, 886, 948, 949, 951, 952, 953, **977**, 986, 987, 989, 991, 1006, 1007, 1009, 1102, 1131, 1194, 1195, 1206

prepared statement not a cursor specification • 986, 1012, 1117

PRESERVE • 56, *144*, *559*, 567, 592

PRIMARY • 52, 67, 71, *146*, 564, *581*, 582, 584, 606, 1251, 1252

<primary datetime field> • 35, 38, 156, 157, 176, 178, 179, 213, 222, 223, 224, 225, *254*, 256, 257, 258, 285, 286, 287, 292, 293, 355, 356, 400, 401, 430, 454, **497**, 498, 499, 1198

PRIOR • *144*, *843*, 889, 890
<privilege column list> • 758, **761**, 762, 763, 769, 770, 1150, 1182, 1226
<privilege method list> • 118, 119, 758, **761**, 762, 763, 764, 769, 770, 1157
privilege not granted • 758
privilege not granted • 759, 1121
privilege not revoked • 785, 1121
PRIVILEGES • *144*, 611, 759, *761*, 762, 785
<privileges> • *758*, 759, **761**, 762, 763, *769*, 770, 785, 786, 1135, 1136, 1155, 1157, 1161, 1227
PROCEDURE • ?, *146*, 525, *529*, 530, 531, *699*, 704, *795*, *1011*
<procedure name> • 83, **160**, *795*, 796, 798, 1034, 1035, 1215
prohibited SQL-statement attempted • 514, 515, 518, 1120
prohibited SQL-statement attempted • 1118
prohibited statement encountered during trigger execution • 917, 1119
PUBLIC • 117, 120, *144*, 166, 527, 635, *761*, 762, 771, 772, 773, 881, 946

— Q —

<qualified asterisk> • **361**, 364, 1225
<qualified identifier> • 87, *159*, **160**, *161*, 162, 163, 164, 165, 196, 198, *199*, 236, *238*, 283, 325, 362, 370, 439, *504*, 505, 510, 517, *660*, 661, 662, 663, 666, 668, 677, 681, 684, 685, 687, 689, 693, 694, 708, 709, 880, 1106, 1107, 1183
<qualified join> • **330**, 331, 374
<quantified comparison predicate> • 23, 29, 248, 298, 367, *393*, 394, **421**, 422, 466, 473, 983, 1225
<quantified comparison predicate part 2> • *209*, **421**
<quantifier> • 298, **421**, 422
<query expression> • v, ?, ?, ?, ?, ?, ?, ?, 54, 55, 56, 57, 58, 65, 69, 73, 76, 77, 88, 108, 110, 202, 204, 227, 249, 298, *302*, 303, *308*, 309, 325, 326, 327, 328, 331, 333, 339, 349, 350, 353, 364, 365, 366, 368, 369, **371**, 372, 373, 374, 375, 376, 378, 379, 380, 381, 382, 383, 384, 385, 390, 391, 453, 538, 548, 550, 551, 568, 571, 592, 593, 596, 601, 602, 604, 610, *613*, 614, 615, 618, 619, 621, 622, 633, 637, 641, 646, 648, 679, 696, 697, 724, 727, 731, 732, 736, 737, 754, 756, 771, 772, 774, 777, 778, 780, 781, 782, 817, *837*, 839, 842, 856, *857*, 859, 861, 868, 874, 879, 900, 902, 903, 905, 912, 982, 1014, 1023, 1027, 1093, 1139, 1148, 1151, 1152, 1153, 1157, 1178, 1182, 1213, 1230, 1233
<query expression body> • 57, 166, 325, 353, **371**, 372, 373, 374, 375, 376, 377, 378, 381, 385, 469, 839, 898, 903, 909, 1212
query expression too long for information schema • 621, 1121
<query name> • 54, 55, 57, 77, **160**, 166, 168, *322*, 325, 326, 327, 328, 329, *371*, 372, 373, 374, 376, 380, 381, 385, 549, 1178
<query primary> • 77, **371**, 374, 376, 377, 379, 839, 898, 903, 909
<query specification> • ?, ?, v, ?, ?, ?, 23, 29, 54, 58, 61, 62, 65, 76, 99, 199, 202, 204, 206, 227, 228, 249, 325, 333, 341, 353, 354, **361**, 362, 364, 365, 366, 367, 368, 369, *371*, 374, 375, 376, 378, 381, 385, 423, 469, 548, 607, 615, 616, 650, 817, 839, 840, 849, 898, 903, 909, *1017*, 1153, 1186, 1212, 1234
<query term> • 77, **371**, 374, 375, 376, 377, 381, 384, 469, 839, 898, 909, 1139, 1212
<question mark> • 23, 83, *138*, **139**, *188*, *411*, 413, 414
<quote> • *137*, **138**, *142*, *144*, 147, 148, *150*, *151*, *152*, 153, 154, 155, 158, 1138, 1172, 1223
<quote symbol> • 148, **150**, 155

— R —

RANGE • 60, *146*, 205, *351*, 353, 355
RANK • 62, 63, 65, *146*, *204*, 205, 538, 1202
<rank function type> • **204**, 366, *536*, 542
READ • ?, ?, ?, ?, 99, 122, 124, 125, *144*, *837*, 839, 888, 924, *925*, 944
read-only SQL-transaction • 855, 859, 865, 872, 892, 894, 1092, 1119
reading SQL-data not permitted • 513, 515, 518, 1118, 1120
READS • *146*, *701*, 709, 716, 720, 726, 1203
REAL • 14, 15, 30, *146*, *172*, 176, 180, 458, 462, 799, 807, 809, 812, 824, 825, 827, 828, 829, 831, 958, 960, 1049, 1058, *1066*, 1070, *1072*, 1073, *1077*, 1081, 1196, 1197
RECURSIVE • 77, *146*, 168, 323, 329, *371*, 372, 383, 384, *613*, 614, 615, 621, 1177, 1178
<recursive search order> • **385**, 386, 1219
REF • ?, ?, 8, 13, 46, 47, 57, *146*, *173*, 181, 459, *559*, 563, 600, 601, 615, 616, 617, 618, *659*, 671, 672, 698, 824, 826, 827, 828, 830, 831, 860, 867, 958, 960, 1058
<reference column list> • **583**, 584, 604
<reference generation> • ?, *559*, **560**, 568, 1159
<reference resolution> • 46, *185*, 186, **243**, 244, 367, 601, 610, 620, 622, 754, 755, 775, 776, 778, 779, 782, 1159
<reference type> • ?, *171*, **173**, 176, 177, 181, 182, *230*, 231, 565, 675, 729, 730, 995, *1046*, 1049, *1054*, 1057, 1058, *1061*, 1064, *1067*, 1070, *1073*, 1075, 1076, *1078*, 1080, *1084*, 1087, 1158, 1162, 1163, 1169
<reference type specification> • *658*, **659**, 660, 661, 673, 1158, 1160
<reference value expression> • *240*, *243*, **247**, 249, 250, 601, 610, 622, 1158
<referenceable view specification> • **613**, 614, 615, 618, 621, 1160

<referenced table and columns> • 67, **583**, 584, 585, 586, 604
<referenced type> • **173**, 176, 177, 181, 675, 1158
REFERENCES • 118, 119, *146*, 200, 327, 567, 568, *583*, 585, 596, 604, 605, 620, 626, 753, 755, 756, 759, *761*, 762, 763, 770, 771, 772, 774, 775, 776, 777, 779, 881, 1251, 1252
<references specification> • ?, *570*, 572, **583**, 590, 1151, 1229
REFERENCING • *146*, *653*, 654
<referencing columns> • 67, **583**, 584, 585, 591, 1180
<referential action> • ?, ?, **583**, 585, 588, 590, 1180, 1251, 1252
<referential constraint definition> • 23, 67, 68, 469, 475, *579*, **583**, 585, 1235, 1252
<referential triggered action> • 67, **583**, 586
<regex capture group> • *253*, **254**, 256, 262, *270*, 272, 273, 278, 985
<regex like predicate> • 23, *393*, 394, **417**, 418, 985, 1154, 1155
<regex like predicate part 2> • *208*, **417**
<regex occurrence> • *253*, **254**, 256, 262, *270*, *271*, 272, 273, 278, 281, 985
<regex occurrences function> • **253**, 255, 262, 264, 985, 1154, 1198, 1211
<regex position expression> • 23, **253**, 255, 256, 262, 263, 264, 985, 1154, 1198, 1211
<regex position start or after> • *253*, **254**, 256, 263
<regex subject string> • **253**, 255, 256, 262, *270*, *271*, 273, 274, 278, 281, 985
<regex substring function> • 21, **270**, 272, 273, 277, 278, 279, 284, 985, 1154, 1211
<regex transliteration> • 21, *270*, **271**, 272, 274, 278, 281, 282, 284, 985, 1154, 1198, 1211
<regex transliteration occurrence> • **271**, 274, 281, 282
REGR_AVGX • 65, *146*, *536*, 541
REGR_AVGY • 65, *146*, *536*, 541
REGR_COUNT • 64, *146*, *536*, 538, 541, 1202
REGR_INTERCEPT • 64, *146*, *536*, 542
REGR_R2 • *146*
REGR_SLOPE • 64, *146*, *536*, 542
REGR_SXX • 64, *146*, *536*, 541
REGR_SXY • 65, *146*, *536*, 541
REGR_SYY • 65, *146*, *536*, 541
<regular character set> • *411*, **412**
<regular character set identifier> • **412**, 413
<regular expression> • 279, **411**, 413, 414
<regular expression substring function> • 20, 21, **270**, 272, 273, 277, 279, 280, 284, 1188
<regular factor> • **411**
<regular identifier> • 24, **141**, 147, 148, 149, *159*, 1141, 1191, 1195, 1224
<regular primary> • **411**
<regular term> • **411**
<regular view specification> • **613**, 618
RELATIVE • ?, 7, 41, 44, *144*, 396, 398, *733*, 735, *843*, 889, 890, 1209
<relative category> • **733**, 734
<relative function specification> • **733**, 734
RELEASE • *146*, *930*
<release savepoint statement> • 87, 106, 122, 514, 518, 709, *816*, **930**, 1102, 1181, 1203
<repeat argument> • 58, **321**, 322, 328, 1193
<repeat factor> • **411**, 414
REPEATABLE • 122, 124, 125, *144*, *321*, 924, *925*
<repeatable clause> • 58, **321**, 328, 1193
<representation> • **658**, 660, 1236
request failed • 855, 860, 865, 872, 892, 894, 1114
request rejected • 887, 1114
<reserved word> • *144*, **145**, 148, 162, 1219, 1221, 1222, 1223, 1245
RESTART • *144*, *751*
RESTRICT • ?, *144*, *556*, *583*, 587, 588, 589, 590, 601, 604, 607, 610, 611, 622, 633, 641, 650, *679*, *692*, 696, *724*, 727, 731, 736, 744, 746, 752, 783, 1180
restrict violation • 587, 588, 589, 590, 1118
restricted data type attribute violation • ?, 999, 1000, 1002, 1003, 1004, 1117
RESULT • 8, 42, *146*, *659*, 661, 663, 664, 665, 672, 681, 683, 684, 688, *699*, *700*, 704, 706, 707, 717, 719, 721, 1156, 1157
<result> • *208*, **209**, 210, 211
<result cast> • 91, 520, 521, 664, 665, 666, 682, 683, 687, **700**, 702, 703, 704, 706, 710, 718, 720
<result cast from type> • 40, 42, 665, 669, 670, 672, 673, 682, 685, 690, **700**, 705, 706, 710, 1187
<result expression> • **209**, 210, 982
<result set cursor> • 97, 99, **1011**, 1012, 1013, 1186
result sets returned • 524, 1121
<result using clause> • 1001, **1006**, 1007, 1016, 1132
RETURN • 10, 92, 98, 100, *146*, 670, 671, 672, 735, *835*, 886, 887, 888, 891, *920*, 1013
<return statement> • 106, 110, 111, 112, 114, 115, 515, *816*, **920**, 1102, 1237
<return value> • **920**, 1000, 1004
<returned result sets characteristic> • 92, **700**, 704, 721, *724*, 725, 1186
RETURNED_CARDINALITY • *144*, *971*, 973, 1005
RETURNED_LENGTH • *144*, *971*, 973, 1005
RETURNED_OCTET_LENGTH • *144*, *971*, 973, 1005

RETURNED_SQLSTATE • *144*, 525, *1096*, 1106, 1107, 1108, 1109
RETURNS • 87, *146*, 661, 670, 671, 672, *700*, *701*, 735
<returns clause> • *660*, 664, 666, 672, 673, 682, 685, 687, 690, *699*, **700**, 703, 704, 920, 1187
<returns data type> • 40, 42, 90, 467, 473, 520, 521, 637, 641, 663, 664, 665, 667, 668, 669, 670, 672, 673, 681, 682, 683, 685, 688, 689, 690, **700**, 703, 705, 706, 707, 710, 711, 713, 714, 715, 718, 719, 721, 722, 920, 1004, 1163, 1164, 1165, 1166, 1173, 1187
<returns table type> • **700**, 702, 722, 1183
<returns type> • 644, **700**, 702
<reverse solidus> • **139**, 147, 1195
REVOKE • ?, *146*, 605, 611, 634, 638, 642, 646, 698, 728, 752, 768, *769*, 785, 1135
<revoke option extension> • **769**, 785, 786, 1135, 1161
<revoke privilege statement> • **769**, 770, 774, 783, 785, 1102
<revoke role statement> • 768, **769**, 770, 774, 783, 785, 1102, 1184
<revoke statement> • 104, 604, 605, 611, 622, 623, 634, 638, 642, 646, 651, 697, 698, 727, 728, 752, 762, **769**, 770, 773, 774, 783, 785, 786, *816*, 1135, 1136, 1231
RIGHT • 72, 73, *146*, *330*, 331, 333, 334, 336, 374, 1104, 1105
<right arrow> • **143**, *238*
<right brace> • *138*, **139**, *411*
<right bracket> • 23, *138*, **139**, *412*, 413, 415, *1052*, *1077*
<right bracket or trigraph> • **139**, *173*, *188*, *193*, *245*, *302*, *308*, *875*
<right bracket trigraph> • **139**, *143*
<right paren> • ?, 23, **138**, *171*, *172*, *173*, *185*, *188*, 197, *202*, *204*, *208*, *212*, *230*, *232*, 234, *243*, *246*, *253*, *254*, *255*, *270*, *271*, *272*, *288*, *290*, *295*, *296*, *302*, *307*, *308*, *311*, *321*, *322*, *330*, *340*, *351*, *361*, *371*, 376, *390*, *403*, *411*, 413, 415, *439*, *497*, *504*, *529*, *535*, *536*, *559*, *560*, *570*, *581*, *583*, *592*, *613*, 619, *648*, *653*, *658*, *659*, *699*, *700*, *729*, *731*, *738*, *741*, *742*, *744*, *761*, *795*, *852*, *857*, *862*, *875*, 979, 980, *1030*, 1031, *1045*, *1046*, *1053*, *1054*, *1060*, *1061*, *1066*, *1067*, *1072*, *1073*, *1077*, *1083*, *1084*, 1225, 1226
<rights clause> • **701**, 709, 722, 1183
ROLE • *144*, *765*, *768*, *946*
<role definition> • 104, 120, *554*, **765**, *815*, 1102, 1184, 1203
<role granted> • **766**, 767
<role name> • 119, *159*, **161**, 167, 168, *765*, *766*, 767, *768*, *769*, 771, 773, 946, 1183
<role revoked> • **769**, 770
<role specification> • **946**, 984
ROLLBACK • 123, 125, *146*, *933*
<rollback statement> • ?, ?, ?, ?, ?, 67, 87, 95, 99, 106, 108, 110, 111, 121, 122, 124, 125, 126, 514, 518, 709, 792, *816*, **933**, 934, 940, 1090, 1102, 1181, 1194, 1203, 1215, 1230
ROLLUP • 62, 70, 76, *146*, *340*, 342
<rollup list> • **340**, 341, 342, 344, 348, 1185
ROUTINE • *144*, *529*, 530, 531, 557, 602, 608, 612, 624, 651, 697, 728, 732, 737, 745, 747, 748, 785
<routine body> • 86, 87, 88, 89, *699*, **701**, 707, 717, 756, 780, 920
<routine characteristic> • **700**, 722, 723, 1134, 1135
<routine characteristics> • *699*, **700**, 704, 722, 1182
<routine invocation> • ?, ?, ?, ?, ?, ?, 79, 82, 87, 88, 89, 92, 93, 128, 131, *185*, 186, 209, *232*, 233, *234*, *236*, 241, 249, 289, 322, 339, 366, 374, 380, 460, **504**, 522, 523, 526, 545, 554, 571, 593, 619, 620, 648, 655, 679, 697, 709, 717, 719, 720, 722, 724, 727, 753, 755, 757, 775, 776, 777, 779, 780, 790, 817, 854, 864, 871, *919*, 952, 984, 986, 995, 1132, 1143, 1144, 1190, 1203, 1236, 1251
<routine name> • 82, 87, 88, 89, 92, 93, 128, 131, 195, 196, 198, 362, 370, 398, 399, **504**, 505, 510, 554, 555, 684, 685, 687, 689, 690, 693, 694, 708, 790, 952, 1037, 1108, 1109, 1183
<routine type> • **529**, 530, 531, 1156
ROUTINE_CATALOG • *144*, *1096*, 1108, 1109
ROUTINE_NAME • *144*, *1096*, 1108, 1109
ROUTINE_SCHEMA • *144*, *1096*, 1108, 1109
ROW • ?, ?, ?, ?, ?, ?, ?, ?, 13, *146*, *173*, 205, 217, 308, *311*, 312, 313, *352*, 353, 356, 358, 359, 386, 387, 459, 478, *653*, 654, 655, 656, 663, 683, 702, 707, 824, 826, 827, 828, 830, 831, 876, 959, 960, 972, 974, 995, 998, 999, 1004, 1176, 1180
<row subquery> • ?, 58, *311*, 312, 313, **390**, 391, 1147
<row type> • *171*, **173**, 177, 181, 182, 183, 1175
<row type body> • **173**
<row value constructor> • 58, **311**, 312, 313, *314*, 315, 436, 1211
<row value constructor element> • 71, 297, **311**, 312, 313, 405, 983, 1147
<row value constructor element list> • **311**, 1211
<row value constructor predicand> • 71, 211, 297, **312**, 313, *314*, 315, 405, 412, 417, 425, 434, 438, 439, 983, 1138, 1172
<row value expression> • 228, *247*, 249, **314**, 315, 316, *403*, 404, 466, 471, 478, 982, 1146, 1176, 1236, 1237
<row value expression list> • **316**
<row value predicand> • 71, *208*, 210, 211, 248, 297, **314**, 315, 367, *395*, *402*, *403*, *405*, 409, *411*, 412, *417*, *419*, *421*, *425*, *427*, *430*, *432*, *434*, *436*, *438*, *439*, 440, 466, 467, 473, 983, 984, 1138, 1171
<row value predicand 1> • **430**, 431, 983
<row value predicand 2> • **430**, 431, 983

<row value predicand 3> • **432**

<row value predicand 4> • **432**

<row value special case> • **314**, 315, 858, 863, 1176

ROW_COUNT • *144*, *1095*, 1104, 1105, 1215

ROW_NUMBER • 63, *146*, *204*, 205, 207, 249, 366, 544, 1188

ROWS • 56, 60, *146*, 205, 249, *351*, 353, 358, *559*, 565, 567, 584, 592, *880*, 985

— S —

S • *1061*

Feature S023, “Basic structured types” • 168, 181, 233, 237, 525, 672, 673, 675, 721, 763, 1155

Feature S024, “Enhanced structured types” • 235, 467, 470, 472, 474, 531, 672, 673, 675, 676, 721, 728, 760, 763, 764, 861, 868, 879, 1156, 1157

Feature S025, “Final structured types” • 673, 1157, 1158

Feature S026, “Self-referencing structured types” • 675, 1158

Feature S027, “Create method by specific method name” • 722, 1158

Feature S028, “Permutable UDT options list” • 673, 1158

Feature S041, “Basic reference types” • 181, 239, 240, 250, 1050, 1059, 1065, 1070, 1076, 1081, 1088, 1158, 1159

Feature S043, “Enhanced reference types” • 182, 226, 242, 244, 568, 600, 602, 621, 673, 861, 1159, 1160

Feature S051, “Create table of type” • 568, 1160

Feature S071, “SQL paths in function and type name resolution” • 191, 503, 555, 578, 792, 952, 1040, 1160, 1161

Feature S081, “Subtables” • 568, 760, 763, 786, 1161

Feature S091, “Basic array support” • 182, 194, 245, 263, 301, 303, 329, 879, 1161, 1162, 1169

Feature S092, “Arrays of user-defined types” • 182, 1162

Feature S094, “Arrays of reference types” • 182, 1162

Feature S095, “Array constructors by query” • 303, 1162

Feature S096, “Optional array bounds” • 182, 1162

Feature S097, “Array element assignment” • 191, 1162

Feature S111, “ONLY in query expressions” • 329, 853, 1162, 1163

Feature S151, “Type predicate” • 440, 1163

Feature S161, “Subtype treatment” • ?, ?, 231, 1163

Feature S162, “Subtype treatment for references” • ?, ?, 231, 1163

Feature S201, “SQL-invoked routines on arrays” • 525, 721, 1163

Feature S202, “SQL-invoked routines on multisets” • 526, 721, 1163

Feature S211, “User-defined cast functions” • 730, 732, 1163, 1164

Feature S231, “Structured type locators” • 721, 797, 1051, 1059, 1065, 1071, 1076, 1081, 1088, 1164, 1165

Feature S232, “Array locators” • 721, 722, 797, 1050, 1059, 1065, 1071, 1076, 1081, 1088, 1165, 1166

Feature S233, “Multiset locators” • 722, 797, 1050, 1059, 1065, 1071, 1076, 1081, 1088, 1166

Feature S241, “Transform functions” • 191, 721, 740, 748, 792, 953, 1040, 1050, 1059, 1065, 1071, 1076, 1081, 1088, 1166, 1167, 1168

Feature S242, “Alter transform statement” • 741, 1168

Feature S251, “User-defined orderings” • 735, 737, 1168

Feature S261, “Specific type method” • 284, 1168

Feature S271, “Basic multiset support” • 182, 194, 246, 263, 307, 309, 329, 435, 438, 545, 1161, 1162, 1168, 1169

Feature S272, “Multisets of user-defined types” • 182, 1169

Feature S274, “Multisets of reference types” • 182, 1169

Feature S275, “Advanced multiset support” • 306, 437, 468, 545, 1169, 1170

Feature S281, “Nested collection types” • 182, 1170

Feature S291, “Unique constraint on entire row” • 582, 1170

<sample clause> • 58, **321**, 327, 328, 329, 1189, 1193

<sample method> • 58, **321**, 328

<sample percentage> • 58, **321**, 328

SAVEPOINT • 86, *146*, *700*, 704, 718, *929*, *930*, *933*

<savepoint clause> • ?, ?, 87, 95, 514, 518, 709, **933**, 934, 1181, 1203

savepoint exception • 929, 930, 934, 1120

<savepoint level indication> • **700**, 704, 722, 1182

<savepoint name> • 121, **161**, 167, 168, *929*, 930, 934, 1181

<savepoint specifier> • ?, ?, 122, **929**, *930*, *933*, 1194

<savepoint statement> • ?, ?, ?, 87, 106, 108, 110, 111, 121, 122, 514, 518, 709, *816*, **929**, 1102, 1181, 1203, 1205

<scalar subquery> • ?, *185*, 186, 240, 246, **390**, 391, 403, 543, 544, 574, 580, 581, 585, 775, 776, 778, 779, 781, 1235

SCALE • *144*, 958, 959, *971*, 974, 975, 976, 995, 1206, 1214

<scale> • 83, *172*, **173**, 175, 176, 806, 822, *1072*, 1073, *1083*, 1087, 1196

SCHEMA • *144*, *553*, *556*, *789*, 790, *949*, *1030*, 1034, 1206

schema and data statement mixing not supported • 818, 1092, 1119

<schema authorization identifier> • 117, **553**, 554

<schema character set or path> • **553**

<schema character set specification> • 183, **553**, 554, 555, 572, 784, 1145, 1202

<schema definition> • 79, 93, 103, 117, 162, 163, 164, 165, 227, 243, 506, 507, 527, **553**, 554, 555, 561, 572, 614, 617, 625, 635, 636, 639, 643, 644, 648, 654, 660, 702, 705, 709, 716, 717, 750, 784, *815*, 1102, 1202, 1230, 1231, 1235

<schema element> • **553**, 555, 1235

<schema function> • **699**

<schema name> • ?, ?, ?, ?, ?, ?, ?, ?, ?, 54, 55, 79, 81, 89, 92, 93, 128, **159**, *160*, *161*, 162, 163, 164, 165, 167, 186, 190, 232, 234, 236, 398, *503*, *504*, 506, 507, 509, 527, 528, 529, *553*, 554, 555, *556*, 557, 561, 564, 567, 572, 579, 594, 611, 614, 616, 617, 623, 625, 626, 628, 633, 635, 637, 639, 641, 643, 644, 646, 648, 651, 654, 655, 657, *660*, 663, 671, 672, 681, 682, 687, 692, 697, 702, 703, 704, 705, 716, 724, 725, 727, 728, 731, 732, 734, 735, 736, 750, 752, *789*, 790, 880, 881, 948, 949, *1030*, 1034, 1106, 1107, 1108, 1109, 1137, 1194, 1195, 1204, 1206, 1209

<schema name characteristic> • **949**, 984

<schema name clause> • 165, **553**, 554, 555, 1137, 1195

<schema name list> • 190, **503**, 554, 720, 790, 952, 1202, 1204

<schema path specification> • 79, **553**, 554, 555, 1160, 1202

<schema procedure> • **699**

<schema qualified name> • ?, ?, 55, 79, 128, **159**, *160*, *161*, 164, 165, 517, 949, 1194, 1195

<schema qualified routine name> • 86, **160**, 166, *529*, 530, 531, 677, *699*, 702, 705, 707, 708, 716, 717, 721, 1183

<schema routine> • ?, ?, 104, *554*, **699**, 702, 721, 1102, 1183

<schema-resolved user-defined type name> • **161**, 163, 177, *529*, 530, 531, *658*, 660, 675, *676*, 677, 679, 681, 687, 689, 692, *696*, *700*, 703, 729, 730, *733*, *736*, *738*, *741*, 742, 744, *746*, *761*, 762, 763, 1155, 1157, 1158

SCHEMA_NAME • *144*, *1096*, 1106, 1107, 1108

SCOPE • *146*, *173*, 455, 564, *601*, 618

<scope clause> • **173**, 177, 181, 182, 213, *560*, 565, 566, 568, *600*, *613*, 617, 1159

<scope option> • *161*, **162**, 166, 167, 168, 838, 1012, 1013, *1025*

SCOPE_CATALOG • *144*, 958, 960, *971*, 974, 976, 995

SCOPE_NAME • *144*, 958, 960, *971*, 974, 976, 995

SCOPE_SCHEMA • *144*, 958, 960, *971*, 974, 976, 995

SCROLL • 92, 98, 100, *146*, *835*, 839, 843, 888, 889, 1013

SEARCH • *146*, *385*

<search clause> • **385**, 386, 387, 388

<search condition> • 33, 52, 59, 63, 66, 68, 69, 71, 72, 73, 75, 76, 123, 200, 202, *208*, 211, 227, 267, 280, 325, 327, *330*, 331, 333, 339, 347, 349, 350, **441**, *535*, 539, 572, 574, 580, 581, 582, 585, *592*, 593, 596, 601, 602, 604, 607, 610, 622, 625, 626, 631, 633, 637, 641, 646, *648*, 649, 650, 651, *653*, 679, 696, 697, 724, 727, 731, 732, 736, 737, 753, 772, 775, 776, 777, 778, 779, 781, 782, 820, *854*, 855, 856, *862*, 863, 864, 865, 867, 868, *871*, 872, 873, 874, 916, 917, 1104, 1105, 1148, 1152, 1153, 1182, 1211, 1226

search condition too long for information schema • 593, 649, 1121

<search or cycle clause> • *371*, 375, **385**, 389, 1219

<searched case> • **208**, 210, 1235

<searched when clause> • **208**, 210, 211

SECOND • 35, 38, *146*, 156, 178, 256, 257, 454, *497*, 498, 499, 576, 976, 981, 985, 1198

<seconds fraction> • **153**, 155, 156, 157, 158, 499, 1146

<seconds integer value> • **153**, 156, 499

<seconds value> • **152**, 156

SECTION • *145*, *1030*

SECURITY • *145*, 177, 200, 214, 215, 241, 277, 327, 528, 532, 618, *701*, 709, 716, 717, 718, 719, 720, 721, 853, 854, 859, 864, 870, 871, 1019, 1021, 1024, 1028

SELECT • ?, ?, ?, 58, 73, 76, 118, 119, *146*, 200, 206, 216, 240, 243, 244, 246, 304, 305, 307, 308, 323, 324, 327, 328, 331, 332, 336, 337, 342, 346, 347, *361*, 365, 375, 377, 382, 386, 387, 388, 543, 544, 566, 567, 568, 574, 580, 581, 582, 585, 596, 604, 605, 614, 619, 620, 621, 655, 686, 753, 754, 755, 756, 758, 759, *761*, 762, 763, 769, 772, 774, 775, 776, 777, 778, 779, 780, 781, 782, 840, *848*, 855, 864, 865, 867, 871, 872, 881, 902, 905, 912, 1104, 1105, 1182, 1252

<select list> • 23, 29, 58, 62, 73, 76, 195, 200, 202, 204, 205, 206, 227, 325, 331, 332, 336, 337, 339, 341, 345, 353, 354, **361**, 362, 364, 365, 366, 367, 369, 374, 377, 386, 423, 607, 617, 650, 777, 778, 779, 780, 782, 839, *848*, 849, 850, 901, 903, 910, 991, 992, 1002, 1182, 1186, 1224, 1228, 1234

<select statement: single row> • ?, 51, 61, 62, 105, 109, 112, 114, 115, 204, 206, 227, 325, 353, 354, 777, 778, 779, 780, 782, *816*, 817, 820, **848**, 849, 1103, 1213, 1234

<select sublist> • 274, **361**, 363, 617, 901

<select target list> • 206, **848**, 849, 850, 851, 1213

SELF • 42, *145*, *659*, 661, 663, 664, 665, 666, 668, 669, 670, 672, 681, 682, 683, 684, 689, 1156

<self-referencing column name> • *559*, **560**, 564, 618

<self-referencing column specification> • **559**, 563, *613*, 615, 616

<semicolon> • *138*, **139**, *653*, *795*, *1030*, 1031, *1052*, *1072*, *1077*, *1083*, *1089*

SENSITIVE • 92, 98, 100, *146*, *835*, 836, 1181

SEPARATE • *1061*

<separator> • **143**, 146, 147, *150*, *151*, 153, 154, 155, 251, 876, 1031, 1042, 1067, 1245

SEQUENCE • *145*, 557, *749*, *751*, *752*, *761*

<sequence column> • **385**, 386

<sequence generator cycle option> • 492, 493, 494, *749*, **750**

<sequence generator data type option> • **749**, 750, 1203

<sequence generator definition> • 104, *554*, **749**, 750, *815*, 1103, 1180, 1203

<sequence generator increment> • 492, 494, **749**

<sequence generator increment by option> • 492, 494, **749**, 750

sequence generator limit exceeded • 491, 1116

<sequence generator max value> • 492, 494, **749**, 1200

<sequence generator maxvalue option> • 492, 494, **749**, 750, 1200

<sequence generator min value> • 492, 494, *749*, **750**, 1200

<sequence generator minvalue option> • 492, 494, **749**, 750, 1200

<sequence generator name> • **161**, 167, 169, *227*, 228, *749*, 750, *751*, *752*, *761*, 762, 1179

<sequence generator option> • **749**

<sequence generator options> • **749**

<sequence generator restart value> • 495, **751**

<sequence generator start value> • 492, **749**

<sequence generator start with option> • 492, **749**, 750

SERIALIZABLE • ?, ?, 65, 122, 123, 124, 125, *145*, 924, *925*, 926, 1091, 1137

serialization failure • 124, 1120

SERVER_NAME • *145*, *1096*, 1109

SESSION • ?, ?, ?, *145*, *943*, *945*

<session characteristic> • ?, ?, ?, **943**

<session characteristic list> • ?, ?, **943**

<session transaction characteristics> • ?, ?, ?, ?, ?, **943**

SESSION_USER • ?, 116, *146*, *187*, 188, 190, 191, 248, *575*, 576, 577, 578, 885, 1093, 1139, 1197

SET • ?, ?, ?, ?, ?, ?, 17, 62, 125, 126, 132, *146*, *171*, 174, 175, *307*, *385*, *438*, 527, *553*, 557, *583*, 585, 587, 588, 589, 590, 595, *598*, *603*, *629*, *635*, *637*, 638, *761*, 806, 819, 820, *862*, *869*, *871*, 921, *923*, 924, *927*, 931, *938*, *943*, *945*, *946*, *947*, *948*, *949*, *951*, *952*, *953*, *954*, *973*, 985, *1021*, *1027*, 1038, *1045*, *1046*, 1047, *1052*, *1053*, 1055, 1058, *1060*, *1061*, 1062, *1066*, 1067, *1072*, 1073, *1077*, 1078, 1081, *1083*, 1084, 1085, 1091, 1181, 1196, 1251, 1252

<set catalog statement> • 84, 107, 128, *817*, **948**, 1103, 1148, 1151

<set clause> • 866, 873, **875**, 876, 877, 878, 879, 895, 984, 1153, 1157

<set clause list> • 325, *862*, 863, 864, 866, *869*, 870, *871*, 873, **875**, 876, 877, *1021*, 1022, *1027*, 1028, 1190

<set column default clause> • *597*, **598**, 1140

<set connection statement> • 106, 126, 127, 135, *816*, 819, **938**, 939, 1091, 1103, 1152

<set constraints mode statement> • ?, 66, 67, 106, *816*, **927**, 928, 1103, 1150, 1251, 1252

<set descriptor information> • **973**

<set descriptor statement> • 84, 107, 109, *817*, 972, **973**, 974, 976, 1103, 1130, 1214

<set domain default clause> • *628*, **629**, 1150

<set function specification> • 76, 77, *185*, 186, 200, **202**, 331, 339, 348, 350, 353, 360, 364, 365, 366, 367, 368, 369, 374, 536, 538, 592, 607, 650, 839, 840, 876, 1143, 1182, 1227, 1234

<set function type> • **535**, 537

<set header information> • **973**, 976

<set item information> • **973**, 974

<set local time zone statement> • 107, 128, *816*, **947**, 984, 1103, 1142

<set names statement> • 84, 107, 129, *817*, **951**, 1103, 1145, 1151

<set path statement> • 84, 107, 128, *817*, **952**, 1103, 1161, 1195

<set predicate> • *393*, 394, **438**, 471, 1169

<set predicate part 2> • *209*, **438**

<set quantifier> • 23, 29, 206, 227, *340*, 341, 348, *361*, 362, 364, 365, 369, 370, **535**, 536, 537, 544, 840, *848*, 849, 1143, 1153, 1176, 1185, 1212, 1224, 1227

<set role statement> • ?, ?, ?, ?, ?, 107, 117, *816*, **946**, 1103, 1184

<set schema statement> • 84, 107, 128, *817*, **949**, 950, 1103, 1151

<set session characteristics statement> • ?, ?, ?, ?, 107, 122, 123, 129, *817*, **943**, 944, 1103, 1151

<set session collation statement> • 107, 129, *817*, **954**, 955, 1103, 1149, 1206

<set session user identifier statement> • 107, 117, *816*, **945**, 984, 1103, 1140, 1205

<set target> • **875**, 876, 984

<set target list> • **875**, 876, 984

<set time zone value> • **947**

<set transaction statement> • 96, 106, 122, 123, *816*, 921, 922, **923**, 924, 1091, 1103, 1181, 1230

<set transform group statement> • 107, 129, *817*, **953**, 978, 1103, 1167

SETS • 76, *145*, *340*, 342, 343, 344, 345, 348, *700*, 704, 1126, 1185

SIGN • *1061*

<sign> • **151**, *152*, 157, 219, 221, *251*, *290*, 291, 1221, 1222

<signed integer> • 30, **151**

<signed numeric literal> • *150*, **151**, 157, 217, 218, 492, 494, 576, *749*, *750*, *751*, 1200

SIMILAR • 20, 23, *146*, *270*, 279, 280, *411*, 412, 413, 414, 416, 979, 1178

<similar pattern> • **411**, 412, 413, 414, 467, 984

<similar predicate> • 23, *393*, 394, **411**, 412, 413, 416, 466, 467, 984, 1175, 1178, 1199

<similar predicate part 2> • *208*, **411**

SIMPLE • *145*, *427*, *583*, 585, 586, 588

<simple case> • **208**, 209, 210, 211, 982, 1138, 1235

<simple comment> • **143**, 147, 978, 1230

<simple comment introducer> • **143**, 147

<simple Latin letter> • **137**, *159*, 415

<simple Latin lower case letter> • **137**, 140, 149, 415

<simple Latin upper case letter> • **137**, 140, 149, 415, *1041*, 1113, 1207

<simple table> • 77, **371**, 374, 375, 376, 379, 382, 384, 550, 839, 1148, 1212

<simple target specification> • ?, 135, **187**, 189, 190, 191, 460, 461, 959, 960, *970*, 971, *1095*, *1096*, 1099, 1106, 1110, 1111, 1137, 1197

<simple target specification 1> • **970**, 971, 972, 1214

<simple target specification 2> • **970**, 971, 972, 1214

<simple value specification> • 126, *161*, *162*, 166, 168, **187**, *188*, 189, 190, 274, 460, 461, 523, *843*, 844, 845, 850, *875*, 877, 878, 879, 889, *925*, 936, 959, 960, 961, *967*, *970*, *973*, *977*, 978, 986, 1011, *1096*, 1162

<simple value specification 1> • **973**, 976

<simple value specification 2> • **973**, 974

<simple when clause> • **208**, 210, 211, 1138

<single datetime field> • 176, **497**, 498, 499

<single group specification> • **701**, 706, 790, 1032

SIZE • ?, ?, *145*, 921, *925*, 926

SMALLINT • 14, 15, 30, *146*, *172*, 176, 180, 458, 462, 799, 805, 806, 824, 825, 827, 828, 829, 830, 958, 960, 1049, 1058, 1064, 1087, 1196

<solidus> • **138**, 147, *251*, 252, *290*, 980

SOME • 64, *146*, *421*, *535*, 537, 540, 544, 1172

<some> • **421**, 422

<sort key> • 61, 205, 353, 354, 356, 357, 358, 473, 538, 543, **546**, 839, 985, 1228

<sort specification> • 61, 354, 538, 539, 543, 545, **546**, 547, 839, 1144, 1202, 1212

<sort specification list> • 61, 303, *351*, 355, 473, *536*, 538, 543, 545, **546**, *837*, 839, 1144, 1202, 1212, 1219

SOURCE • *145*, *659*

<source character set specification> • **643**, 772

<source data type> • **729**, *731*

SPACE • 19, 25, *145*, 412, 415, 528, *639*

<space> • 10, 18, 19, 85, 126, *137*, **138**, 140, 146, *151*, *152*, 153, 158, 190, 217, 218, 219, 220, 267, 268, 280, 281, 282, 400, 415, 445, 449, 450, 577, 809, 811, 812, 1032, 1067, 1108, 1172

SPECIFIC • *146*, *529*, 557, 602, 608, 612, 624, 651, *660*, 697, *700*, 722, 728, 732, 735, 737, 745, 747, 748, 785, 1158

<specific method name> • 40, 41, **660**, 663, 669, 670, 681, 682, 685, 686, 687, 690, 691, 692, *700*, 702, 703, 704

<specific method specification designator> • **692**

<specific name> • ?, 87, **160**, 167, *529*, 557, 602, 608, 651, 697, *700*, 703, 704, 705, 717, 719, 724, 727, *733*, 734, 745, 747, 748, 785, 1108, 1109

<specific routine designator> • 41, 98, 514, **529**, 530, 531, *643*, 644, *724*, *727*, 728, *729*, *733*, 734, 735, *738*, 760, *761*, 762, 763, 888, *1011*, 1012, 1013, 1156, 1157

<specific type method> • *270*, **271**, 272, 276, 278, 282, 283, 284, 1168

SPECIFIC_NAME • *145*, *1096*, 1108, 1109

SPECIFICTYPE • *146*, *271*, 735

SQL • ?, 9, 87, 88, *146*, 177, 200, 214, 215, 241, 277, 327, 366, *501*, 515, 519, 520, 521, 528, 532, 618, 661, 663, 664, 669, 671, 672, 682, 683, 685, *701*, 704, 706, 709, 716, 717, 718, 719, 720, 725, 726, 735, *738*, 790, 853, 854, 859, 864, 870, 871, *967*, *969*, *970*, *973*, *991*, *1001*, 1019, 1021, 1024, 1028, *1029*, *1030*, *1046*, 1047, 1048, 1049, *1053*, *1054*, 1055, 1056, 1057, *1060*, *1061*, 1062, 1063, 1064, *1066*, *1067*, 1068, 1069, 1070, *1072*, *1073*, 1074, 1075, *1077*, *1078*, 1079, 1080, *1083*, *1084*, 1085, 1086, 1087, 1156, 1203

<SQL argument> • 232, 374, 460, **504**, 505, 506, 507, 508, 523, 525, 526, 985, 986, 1132, 1163

<SQL argument list> • 88, 186, *232*, *234*, *236*, *238*, *241*, 374, 456, 460, **504**, 505, 506, 507, 508, 708, 986

<SQL condition> • **1041**, 1044, 1133, 1145

<SQL connection statement> • 87, 135, 655, 709, *815*, **816**, 817, 818, 819, *1089*, 1090, 1091, 1109, 1201, 1203

<SQL control statement> • 87, 110, *815*, **816**, 817, *977*, 978

<SQL data change statement> • 655, 709, **816**, 1203

<SQL data statement> • ?, 709, *815*, **816**, 818, 1203

<SQL descriptor statement> • **817**

<SQL diagnostics information> • **1095**, 1109

<SQL diagnostics statement> • 96, 135, *815*, **817**, 819, 820, 821

<SQL dynamic data statement> • ?, 121, **817**, 818, 1194

<SQL dynamic statement> • 167, 168, 655, 709, *815*, **817**, 997, 1001, 1203

<SQL executable statement> • **815**

<SQL language character> • 24, **137**, 175, 218, 219, 220, 221, 554, 794, 1032, 1191, 1204

<SQL language identifier> • *141*, **159**, *161*, 162, 165

<SQL language identifier part> • **159**, 162

<SQL language identifier start> • **159**

<SQL parameter declaration> • 86, 87, 91, 637, 641, 663, 664, 666, 682, 684, 689, 690, **699**, 705, 706, 707, 710, 712, 713, 714, 715, 719, 721, 1157

<SQL parameter declaration list> • 40, 41, 42, 195, 196, 362, 530, *660*, 663, 664, 669, 670, 681, 682, 687, 688, **699**, *700*, 703, 705, 706

<SQL parameter name> • 90, **160**, 167, 195, 196, 362, 648, 663, 664, 667, 671, 672, 682, 688, *699*, 703, 706, 707, 1038, 1109

<SQL parameter reference> • 166, *187*, *188*, 189, 190, 197, **201**, 508, 523, 560, 570, 592, 613, 655, 844, 845, 848, 850, 885, 997, 1001

<SQL prefix> • **1029**, *1030*, 1031, 1032

<SQL procedure statement> • 65, 83, 87, 88, 93, 96, 111, 112, 135, 167, 168, 289, 325, 460, 586, 601, *653*, *701*, 709, 717, 756, 757, *795*, 796, 797, **815**, 817, 818, 820, 917, 978, *1029*, 1035, 1036, 1038, 1039, 1042, 1043, 1090, 1203, 1210, 1215, 1252

<SQL routine body> • 607, 650, **701**, 709, 718, 719, 722, 781, 782, 1190, 1201, 1203

SQL routine exception • 513, 514, 515, 518, 1108, 1120

<SQL routine spec> • 177, 200, 214, 215, 241, 277, 327, 528, 532, 618, **701**, 709, 716, 853, 854, 859, 864, 870, 871, 1019, 1021, 1024, 1028

<SQL schema definition statement> • **815**

<SQL schema manipulation statement> • **815**

<SQL schema statement> • 79, 97, 134, 163, 166, 177, 200, 214, 215, 241, 277, 327, 505, 528, 532, 578, 596, 618, 654, 655, 709, 716, **815**, 817, 819, 820, 833, 853, 854, 859, 864, 870, 871, *977*, 1009, 1019, 1021, 1024, 1025, 1028, *1089*, 1092, 1203

<SQL session statement> • 655, *815*, **816**, *978*, *1089*, 1203

<SQL special character> • **137**, *142*

<SQL statement name> • ?, 11, **161**, 169, 519, *977*, 986, *989*, *991*, 992, *1006*, 1130, 1234

<SQL statement variable> • ?, 11, 110, 837, **977**, 978, *1008*

<SQL terminal character> • **137**, 1195

<SQL terminator> • *1029*, **1030**, 1031

<SQL transaction statement> • 87, 655, 709, *815*, **816**, *977*, *1089*, 1201, 1203

<SQL-client module definition> • ?, ?, ?, ?, ?, 55, 81, 82, 83, 85, 89, 97, 98, 111, 112, 117, 154, 162, 163, 164, 165, 167, 168, 188, 189, 199, 506, 507, 512, 513, 554, 566, 617, 626, 635, 636, 639, 644, 648, 654, 655, 668, 676, 702, 716, 717, 720, 750, **789**, 790, 791, 792, 793, 794, 795, 796, 797, 798, 833, 834, 837, 880, 987, 989, 991, 1006, 1007, 1009, 1012, 1013, 1033, 1039, 1040, 1133, 1134, 1193, 1201, 1215, 1234

<SQL-client module name> • 83, 135, **160**, 166, *794*, 798, 1034, 1210, 1215

SQL-client unable to establish SQL-connection • 936, 1114

<SQL-data access indication> • *660*, 664, 672, 682, *700*, **701**, 704, 716, 720, *724*, 725, 726, 1156

<SQL-invoked function> • 86, **699**, 702, 707, 920

<SQL-invoked procedure> • 86, **699**, 702, 707, 1236

<SQL-invoked routine> • 11, 88, 92, 505, 611, 612, 623, 671, 697, **699**, 702, 704, 705, 706, 707, 716, 717, 721, 722, 735, *815*, 817, 986, 989, 991, 1006, 1039, 1187, 1190, 1236

<SQL-path characteristic> • **952**, 984

<SQL-server name> • 126, **161**, 166, *935*, 936, 1109, 1194, 1205

SQL-server rejected establishment of SQL-connection • 936, 1114

SQLEXCEPTION • *146*, *1041*, 1042, 1044

SQLSTATE • 88, 93, 94, 96, *146*, 519, 712, *795*, 796, 806, 807, 808, 822, 823, 825, 826, 828, 829, 830, 1033, 1035, 1036, 1039, *1041*, 1042, 1043, 1044, 1049, 1099, 1106, 1113, 1114, 1122, 1133, 1207, 1210

<SQLSTATE char> • **1041**, 1042

<SQLSTATE class value> • **1041**, 1042, 1043, 1044

<SQLSTATE subclass value> • **1041**, 1042, 1043

SQLWARNING • *146*, *1041*, 1042, 1044

SQRT • *146*, *255*, 537, 541

<square root> • 32, *253*, **255**, 257, 264, 1189

<standard character set name> • **527**, 528

START • *146*, *254*, 256, 263, 562, *749*, *921*, 922, 1181

<start field> • 176, 293, 454, **497**, 498, 499

<start position> • *253*, 256, 262, *270*, *271*, **272**, 273, 274, 275, 277, 278, 281, 283, 985, 1211

<start transaction statement> • 106, 108, 110, 122, 123, *816*, **921**, 922, 1091, 1103, 1181

STATE • 7, 16, 41, 44, *145*, 396, 399, 669, *733*, 735, 737

<state category> • **733**, 734, 735

STATEMENT • *145*, *653*, 655, 656, *1096*, 1110, 1252

statement completion unknown • 126, 1120

<statement cursor> • 97, 99, 838, **1011**, 1012

<statement information> • **1095**, 1110

<statement information item> • **1095**, 1096, 1099, 1110

<statement information item name> • **1095**, 1096, 1097, 1109, 1110, 1111, 1137, 1186

<statement name> • 98, **161**, 168, 837, 986, 989, 991, 1006, 1008, 1009, 1014, 1215

<statement or declaration> • **1029**, 1039

statement too long for information schema • 655, 1121

STATIC • 40, 42, 85, 91, *146*, 506, 519, *529*, 530, 531, *660*, 663, 665, 666, 667, 669, 672, 673, 681, 684, 685, 687, 688, *692*, 693, 694, 695, *700*, 702, 703, 719, 735, *789*, 791, 792, 793, 987, *1030*, 1133, 1156

<static method invocation> • 87, *185*, 186, **234**, 235, 679, 697, 724, 727, 775, 776, 777, 779, 780, 1156

<static method selection> • **234**, 505, 506, 507, 509

<status parameter> • 460, **795**, 796

STDDEV_POP • 64, *146*, *535*, 536, 537, 545, 1189

STDDEV_SAMP • 64, *146*, *535*, 536, 537, 545, 1189

string data, length mismatch • 1116

string data, right truncation • 218, 219, 220, 221, 267, 268, 279, 281, 445, 446, 449, 450, 451, 1109, 1116, 1121

<string length> • *270*, *271*, **272**, 275, 277, 278, 283

<string value expression> • *188*, 189, 190, *247*, 249, *254*, 256, 257, 258, 259, 263, **265**, 267, 277, 284, 467, 979, 1143, 1246

<string value function> • *265*, **270**, 272, 277, 979, 980, 1245

STRUCTURE • *145*, *991*

STYLE • *145*, 366, 515, 517, 519, 520, 521, 664, 682, *700*, 706, 709, 714, 718, 720, 725, 726

SUBCLASS_ORIGIN • *145*, *1096*, 1106, 1122, 1207

SUBMULTISET • *146*, *436*

<submultiset predicate> • 248, 367, *393*, 394, **436**, 437, 471, 1170

<submultiset predicate part 2> • *209*, **436**

<subquery> • ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 111, 227, 368, 369, **390**, *560*, 563, 1230, 1234

SUBSTRING • 20, *146*, 258, *270*, *272*, 275, 276, 277, 979

substring error • 278, 283, 1116

SUBSTRING_REGEX • 21, *146*, *270*, 1219

<subtable clause> • 57, *559*, **560**, 563, 564, 566, 567, 568, 1161

<subtype clause> • 43, **658**, 661, 662, 668, 669, 671

<subtype operand> • **230**

<subtype treatment> • ?, *185*, 186, **230**, 231, 1163

<subview clause> • **613**, 615, 616, 617, 618, 620, 621

successful completion • 93, 94, 477, 525, 821, 1043, 1092, 1113, 1120

SUM • 63, *146*, *535*, 537, 540, 1197, 1201

<supertable clause> • **560**

<supertable name> • **560**, 566

<supertype name> • **658**, 662

SYMMETRIC • *146*, *402*, 1186

syntax error or access rule violation • 820, 889, 978, 985, 987, 1007, 1008, 1012, 1014, 1090, 1107, 1113, 1120

SYSTEM • *146*, *321*, *560*, 563, 568, 615, *659*, *857*, 860, 867, 868, 1159

<system-generated representation> • **659**, 661

SYSTEM_USER • *146*, *187*, 188, 190, 191, 248, 367, *575*, 576, 577, 578, 885, 1093, 1139, 1197

— T —

T • *141*, *145*, 149, 174, 1126, 1175, 1246

Feature T021, “BINARY and VARBINARY data types” • 158, 182, 1050, 1059, 1065, 1071, 1076, 1082, 1088, 1170, 1171, 1172

Feature T022, “Advanced BINARY and VARBINARY data type support” • 284, 409, 1138, 1171, 1175

Feature T023, “Compound binary literals” • 158, 1171, 1172

Feature T024, “Spaces in binary literals” • 158, 1172

Feature T031, “BOOLEAN data type” • 158, 181, 249, 299, 313, 544, 1172

Feature T041, “Basic LOB data type support” • 158, 182, 722, 1050, 1059, 1065, 1071, 1076, 1081, 1082, 1088, 1170, 1172, 1173, 1174

Feature T042, “Extended LOB data type support” • 226, 284, 409, 410, 416, 467, 671, 1143, 1171, 1174, 1175

Feature T043, “Multiplier T” • 149, 1175

Feature T044, “Multiplier P” • 149, 1175

Feature T051, “Row types” • 168, 182, 184, 229, 313, 315, 369, 1175, 1176

Feature T052, “MAX and MIN for row types” • 545, 1176

Feature T053, “Explicit aliases for all-fields reference” • 370, 1176

Feature T061, “UCS support” • 182, 284, 426, 1124, 1176, 1177

Feature T071, “BIGINT data type” • 182, 1050, 1059, 1177

Feature T111, “Updatable joins, unions, and columns” • 368, 379, 380, 614, 621, 841, 856, 861, 868, 874, 1177

Feature T121, “WITH (excluding RECURSIVE) in query expression” • 168, 329, 383, 1177, 1178

Feature T122, “WITH (excluding RECURSIVE) in subquery” • 383, 1178

Feature T131, “Recursive query” • 383, 621, 1178

Feature T132, “Recursive query in subquery” • 384, 1178

Feature T141, “SIMILAR predicate” • 416, 1178

Feature T151, “DISTINCT predicate” • 433, 1178

Feature T152, “DISTINCT predicate with negation” • 433, 1178, 1179

Feature T171, “LIKE clause in table definition” • 568, 1179

Feature T172, “AS subquery clause in table definition” • 569, 1179

Feature T173, “Extended LIKE clause in table definition” • 569, 1179

Feature T174, “Identity columns” • 574, 603, 1179

Feature T175, “Generated columns” • 574, 1179

Feature T176, “Sequence generator support” • 169, 228, 750, 751, 752, 1179, 1180

Feature T191, “Referential action RESTRICT” • 590, 1180

Feature T201, “Comparable data types for referential constraints” • 591, 1180

Feature T211, “Basic trigger capability” • 329, 656, 657, 763, 1180

Feature T212, “Enhanced trigger capability” • 656, 1180

Feature T231, “Sensitive cursors” • 836, 1153, 1180, 1181

Feature T241, “START TRANSACTION statement” • 922, 1181

Feature T251, “SET TRANSACTION statement: LOCAL option” • 924, 1181

Feature T261, “Chained transactions” • 932, 934, 1181

Feature T271, “Savepoints” • 168, 929, 930, 934, 1181

Feature T272, “Enhanced savepoint management” • 514, 518, 718, 722, 1181, 1182
Feature T281, “SELECT privilege with column granularity” • 763, 1182
Feature T301, “Functional dependencies” • 349, 350, 360, 369, 972, 1182
Feature T312, “OVERLAY function” • 284, 1182
Feature T322, “Overloading of SQL-invoked functions and procedures” • 721, 1182, 1183
Feature T323, “Explicit security for external routines” • 721, 1183
Feature T324, “Explicit security for SQL routines” • 722, 1183
Feature T325, “Qualified SQL parameter references” • 198, 370, 1183
Feature T326, “Table functions” • 309, 329, 722, 1183
Feature T331, “Basic roles” • 168, 765, 767, 768, 785, 946, 1183, 1184
Feature T332, “Extended roles” • 191, 578, 763, 765, 787, 1184
Feature T351, “Bracketed comments” • 149, 1184
Feature T431, “Extended grouping capabilities” • 203, 348, 1184, 1185
Feature T432, “Nested and concatenated GROUPING SETS” • 348, 1185
Feature T433, “Multiargument GROUPING function” • 203, 1185
Feature T434, “GROUP BY DISTINCT” • 348, 1185
Feature T441, “ABS and MOD functions” • 263, 1185
Feature T461, “Symmetric BETWEEN predicate” • 402, 1185, 1186
Feature T471, “Result sets return value” • 721, 836, 1013, 1186
Feature T491, “LATERAL derived table” • 329, 1186
Feature T501, “Enhanced EXISTS predicate” • 423, 1186
Feature T511, “Transaction counts” • 1111, 1186
Feature T551, “Optional key words for default syntax” • 384, 836, 1186, 1187
Feature T561, “Holdable locators” • 882, 883, 1187
Feature T571, “Array-returning external SQL-invoked functions” • 672, 721, 1187
Feature T572, “Multiset-returning external SQL-invoked functions” • 673, 721, 1187
Feature T581, “Regular expression substring function” • 284, 1187, 1188
Feature T591, “UNIQUE constraints of possibly null columns” • 582, 1188
Feature T601, “Local cursor references” • 169, 1188
Feature T611, “Elementary OLAP operations” • 207, 360, 547, 1188
Feature T612, “Advanced OLAP operations” • 169, 207, 264, 360, 545, 1188, 1189
Feature T613, “Sampling” • 329, 1189
Feature T621, “Enhanced numeric functions” • 263, 264, 545, 1189, 1190
Feature T641, “Multiple column assignment” • 879, 1190
Feature T651, “SQL-schema statements in SQL routines” • 515, 722, 1190
Feature T652, “SQL-dynamic statements in SQL routines” • 515, 722, 1190
Feature T653, “SQL-schema statements in external routines” • 518, 722, 1190
Feature T654, “SQL-dynamic statements in external routines” • 518, 722, 1190
Feature T655, “Cyclically dependent routines” • 722, 1190
TABLE • ?, ?, ?, ?, ?, *146*, *308*, *322*, *371*, 375, 381, 387, 556, *559*, *594*, 602, 605, 608, *610*, 611, 612, 624, 634, 651, *653*, 654, 655, *700*, 702, 732, 737, *761*, 784, 785, *880*, 881, 1135, 1251, 1252
<table commit action> • **559**, *880*
<table constraint> • **579**, 580, 634, 1108
<table constraint definition> • *559*, 562, 564, 567, 572, 573, **579**, 580, 582, 585, *606*, 634, 1229
<table contents source> • **559**, 560
<table definition> • ?, ?, ?, 54, 57, 103, 162, 164, 227, *553*, **559**, 561, 566, 568, 570, 572, 573, 579, 581, 582, 583, 592, 596, *815*, 1103, 1235
<table element> • **559**, 561, 566
<table element list> • **559**, 563, 565, *880*
<table expression> • 60, 61, 76, 199, 204, 206, **318**, 349, 352, 354, 361, 364, 366, 367, 368, 369, 373, 374, 386, 777, 778, 779, 780, 782, 839, *848*, 849, 898, 903, 909, 1233, 1234
<table factor> • v, 74, 196, **321**, 324, 326, 327, 328, 329, *330*, 331, 373, 374
<table function column list> • **700**, 702
<table function column list element> • **700**
<table function derived table> • *321*, **322**, 329, 1183
<table name> • ?, ?, ?, ?, ?, ?, 54, 57, 79, **159**, 162, 164, 166, *173*, 177, *322*, 325, 326, 327, 328, 548, 556, *559*, *560*, 561, 562, 564, 566, 567, 572, 579, 581, 582, *583*, 585, *594*, 595, 600, 601, 602, 604, 606, 607, 608, *610*, 611, 612, *613*, 614, 616, 617, 618, 619, *622*, 623, 624, 632, *653*, 654, 656, 685, 690, 732, 737, 754, *761*, 762, 763, 769, 770, 773, 777, 778, 781, 783, 784, 785, 839, *852*, 853, 854, *857*, 861, 863, 865, 869, 871, 872, *880*, 931, 1007, 1019, 1021, 1023, 1027, 1032, 1104, 1107, 1152, 1161, 1224, 1231
<table or query name> • 56, 73, 74, 197, 199, *321*, **322**, 324, 325, 326, 327, 328, 363, 367, *371*, 372, 375, 376, 378, 616, 839, 853, 854, 855, 865, 867, 868, 869, 871, 872, 1027, 1152, 1224
<table primary> • v, 65, 74, **321**, 322, 324, 327, 328, 331, 374

<table reference> • v, ?, 58, 65, 72, 74, 75, 199, *319*, 320, **321**, 324, 325, 326, 327, 329, *330*, 331, 367, 368, 373, 374, 380, 386, 548, 592, 614, 616, 619, 754, 771, 772, 774, 775, 776, 777, 779, 780, 781, 782, 852, 855, *862*, 863, 865, 867, 868, 869, 872, 898, 903, 909, 910, 1019, 1021, 1023, 1027, 1104, 1152, 1162, 1193, 1224, 1231, 1233

<table reference list> • 75, **319**, 320, 373

<table row value expression> • **314**, 315, *316*, 317

<table scope> • **559**, 563, 566, 568, 1146

<table subquery> • ?, 248, 298, *321*, *322*, 362, **390**, *403*, *421*, *423*, *424*, *427*, 428, 429, 466, 467, 469, 473, 983, 1186, 1222, 1224, 1225, 1226

<table value constructor> • v, 54, 72, 313, **316**, 317, *371*, 375, 376, 379, 384, 403, 550, 859, 901, 982, 1147, 1148, 1211

<table value constructor by query> • **308**, 309, 1183

TABLE_NAME • *145*, *1096*, 1106, 1107, 1108

TABLESAMPLE • *146*, *321*

<target array element specification> • *187*, **188**, 189, 191, 508, 523, 844, 845, 848, 849, 850, 1162

<target array reference> • **188**, 189

<target character set specification> • **643**, 772

<target data type> • **729**, *731*

<target specification> • ?, ?, ?, 135, **187**, 189, 191, 460, 461, *504*, 508, 523, 820, *843*, 844, 845, *848*, 849, 850, 851, 986, *1001*, 1002, 1003, 1004, 1197, 1236

<target subtype> • **230**, 231, 1163

<target table> • 777, 781, **852**, 853, *854*, 855, 856, 857, *862*, 863, 864, 865, 866, 867, 868, *869*, *871*, 872, 873, 874, 875, 893, 895, *1019*, *1021*, *1023*, *1027*, 1104, 1163, 1177

target table disagrees with cursor specification • 1019, 1021, 1120

TEMPORARY • ?, ?, 54, *145*, *559*, 565, 566, 567, *880*

<temporary table declaration> • ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 55, 81, 105, 110, 112, 114, 115, 127, 162, 199, 512, 513, 571, 573, 592, *789*, **880**, 881, *1029*, 1032, 1035, 1039, *1089*, 1146, 1201

<term> • **251**, *290*, 291

THEN • *146*, 205, *208*, 209, 210, 304, 305, 307, 387, *862*, 1110

TIES • *145*, *352*, 360

TIME • 14, 15, 33, 34, 35, 36, 37, 96, *146*, *151*, 155, 156, *173*, 176, 178, 180, 222, 223, 224, 225, 256, *285*, 286, 288, 289, 293, 333, 365, 366, 459, 462, 464, 824, 826, 827, 828, 829, 831, *947*, 975, 976, 983, 985, 1199

<time fractional seconds precision> • 34, 35, **173**, 176, 178, 180, 257, 485, 822, 1197

<time interval> • **152**

<time literal> • **151**, 155, 157, 158, 1146, 1232

<time precision> • **173**, 176, 178, 181, 222, 224, *288*, 289, 958, 995, 1146, 1197

<time string> • *142*, **151**

<time value> • **152**, 157, 1232

<time zone> • 248, **285**, 286, 287, 1142

<time zone field> • **254**, 256, 263, 1136, 1142

<time zone interval> • **152**, 155, 156, 157, 158, 1142, 1232

<time zone specifier> • **285**, 286

TIMESTAMP • 14, 15, 33, 34, 35, 36, 37, 96, *146*, *151*, 156, *173*, 176, 178, 180, 222, 223, 224, 225, 256, 285, 288, 289, 293, 333, 365, 366, 459, 462, 464, 824, 826, 827, 828, 829, 831, 976, 983, 985

<timestamp literal> • **151**, 156, 157, 158, 1146, 1232

<timestamp precision> • **173**, 176, 178, 181, 223, *288*, 289, 958, 995, 1146, 1197

<timestamp string> • *142*, 146, **151**

TIMEZONE_HOUR • *146*, 178, *254*, 258

TIMEZONE_MINUTE • *146*, 178, *254*

TO • ?, ?, ?, ?, 20, 35, 128, *146*, 225, 279, 280, 286, 292, 293, *385*, 401, *411*, 412, 413, 414, 416, 454, *497*, 498, 527, 568, 621, 635, *643*, 671, *738*, *744*, 745, 754, 755, 756, 757, *758*, 759, *766*, 819, *933*, *935*, 947, 976, 980, 981, 984, 985, *1041*, 1042, 1043, 1044, 1091

<to sql> • **738**, 739, 740, 742, 743, 744

<to sql function> • **738**, 739, 742

<token> • **141**, 146, 147, 1032, 1245

too many • 929, 1120

TOP_LEVEL_COUNT • *145*, *970*, 973, 992

TRAILING • *146*, *271*, 282, 284

TRANSACTION • ?, ?, ?, ?, ?, 125, 126, *145*, *921*, 922, *923*, 924, *943*, 1181

<transaction access mode> • ?, ?, 924, **925**, 943, 944, 1230

<transaction characteristics> • ?, *921*, *923*, 924, **925**, 1137

<transaction mode> • ?, ?, **925**, *943*, 1230

transaction resolution unknown • 126, 1114

transaction rollback • ?, ?, ?, 124, 126, 931, 932, 1106, 1107, 1120, 1122

TRANSACTION_ACTIVE • *145*, *1095*, 1105, 1111, 1186

TRANSACTIONS_COMMITTED • *145*, *1095*, 1105, 1111, 1186

TRANSACTIONS_ROLLED_BACK • *145*, *1095*, 1105, 1111, 1186

<transcoding> • 22, **270**, 272, 273, 274, 277, 282, 284, 1149, 1198

<transcoding name> • **161**, 164, 167, 168, *270*, 274, 282, 1149, 1195

TRANSFORM • *145*, 670, 697, *701*, 728, *738*, *741*, *746*, *953*

<transform definition> • 45, 104, *554*, **738**, 740, *815*, 1103, 1167

<transform element> • **738**

<transform element list> • **738**, 739, *742*

<transform group> • **738**
<transform group characteristic> • **953**, 984
<transform group element> • **746**
<transform group specification> • **701**, 706, 721, *789*, *1030*, 1167
<transform kind> • **744**
TRANSFORMS • *145*, *738*, *741*, *746*
<transforms to be dropped> • **746**
<transition table name> • 54, 55, 57, *322*, 326, 328, 329, 548, **653**, 1180
<transition table or variable> • **653**
<transition table or variable list> • **653**, 654
TRANSLATE • *146*, *270*
TRANSLATE_REGEX • 21, *146*, *271*, 1219
TRANSLATION • *146*, 557, *643*, *646*, 754, *761*
<transliteration definition> • 104, 164, *554*, **643**, 644, 645, 754, 772, *815*, 1103, 1149, 1202
<transliteration name> • 22, **161**, 164, 167, 168, *271*, 274, 277, 281, 557, 626, *643*, 644, *646*, 754, *761*, 762, 772, 1149
<transliteration routine> • **643**, 644
<transliteration source> • **643**, 644
TREAT • ?, *146*, *230*
TRIGGER • 118, 119, *146*, 557, 568, 605, 609, 612, 624, 651, *653*, 655, *657*, 732, 737, *761*, 762, 763, 777, 784, 1180, 1252
<trigger action time> • 197, **653**, 656
<trigger column list> • 135, **653**, 654, 656
<trigger definition> • ?, ?, 55, 104, 132, 134, 135, 197, *554*, **653**, 654, 655, 656, *815*, 1103, 1180, 1203
<trigger event> • 135, **653**, 654, 656
<trigger name> • **160**, 167, 557, *653*, 654, 655, 656, *657*, 732, 737, 784, 1107
TRIGGER_CATALOG • *145*, *1096*, 1107
TRIGGER_NAME • *145*, *1096*, 1107
TRIGGER_SCHEMA • *145*, *1096*, 1107
<triggered action> • 88, 134, 135, 289, 596, 610, 622, **653**, 654, 655, 656, 865, 866, 873, 1180, 1203
triggered action exception • ?, 931, 1107, 1120
triggered action exception • 917, 1107, 1120
triggered data change violation • 589, 590, 1106, 1107, 1120
<triggered SQL statement> • ?, ?, 132, 135, 508, **653**, 655, 777, 778, 779, 854, 859, 871, 916, 931, 1203
TRIM • *146*, 221, 222, 223, 224, 225, *271*, *272*, 275, 277, 935, 936, 945, 946, 948, 949, 951, 952, 953, 954, 967, 986, 1011
<trim character> • **271**, 275, 282
trim error • 282, 283, 1116
<trim function> • 22, 29, *270*, **271**, 272, 275, 278, 282, 1223, 1245
<trim octet> • **272**, 277, 283
<trim operands> • **271**
<trim source> • **271**, 275, 277, 282, 283
<trim specification> • **271**, *272*, 275, 277, 282, 283
TRUE • *146*, *153*, 158, 219, 221, *296*, 298, 387, 399, 735, 812, 854
<truth value> • **296**, 297, 299, 1147
TYPE • ?, ?, *145*, 557, *658*, *676*, *696*, 698, *701*, *761*, 785, *953*, 958, 959, 960, 961, *971*, 972, 974, 975, 976, 993, 994, 995, 998, 999, 1004, 1005, *1046*, 1047, 1048, 1049, *1053*, *1054*, 1055, 1056, 1057, *1060*, *1061*, 1062, 1063, 1064, *1066*, *1067*, 1068, 1069, 1070, *1072*, *1073*, 1074, 1075, *1077*, *1078*, 1079, 1080, *1083*, *1084*, 1085, 1086, 1087, 1206, 1214
<type list> • **439**
<type predicate> • *393*, 394, **439**, 440, 1163
<type predicate part 2> • *209*, **439**
<typed table clause> • **559**, 563, 566, 567
<typed table element> • **559**
<typed table element list> • **559**, 565

— U —

U • *142*, *150*
UCS • 256, 273, 274, 480, 481, 483, 1200
UESCAPE • 20, *142*, *146*, 147
UNBOUNDED • *145*, 205, *352*, 353, 356, 357, 358, 359
UNCOMMITTED • 122, 124, 125, *145*, 924, *925*
undefined DATA value • 972, 1117
UNDER • 44, 58, 118, 119, *145*, *560*, 566, *613*, 618, 620, *658*, 668, 672, 676, 756, *761*, 762, 763, 775, 780, 1155, 1161
<underscore> • 22, 23, *138*, **139**, *150*, *159*, 162, 407, 408, 409, *412*, 413, 415, 805, 1224
<Unicode 4 digit escape value> • **142**, 147
<Unicode 6 digit escape value> • **142**, 147
<Unicode character escape value> • **142**, 148
<Unicode character string literal> • *141*, 146, **150**, 153, 154, 158, 1141
<Unicode delimited identifier> • *141*, **142**, 146, 147, 148, 149, *159*, 1141
<Unicode delimiter body> • **142**, 147, 148
<Unicode escape character> • **142**, 147, 148, 1195
<Unicode escape specifier> • **142**, 147, *150*
<Unicode escape value> • **142**, 147, *150*, 154
<Unicode identifier part> • **142**, 147
<Unicode representation> • **150**, 153
UNION • ?, 23, 29, 50, 69, 77, *146*, 244, 248, 249, *304*, 305, 306, 323, 336, 347, *371*, 373, 375, 377, 379, 380, 381, 382, 383, 384, 400, 469, 471, 543, 621, 898, 909, 1170, 1186, 1212, 1231, 1235

UNIQUE • 57, 67, *146*, *424*, *427*, 428, 429, 564, *581*, 582, 1139, 1170, 1188
<unique column list> • 67, 469, 564, **581**, 582, 584, 1188, 1229, 1230
<unique constraint definition> • 23, 29, 469, 564, *579*, **581**, 582, 584, 604, 1229, 1230
<unique predicate> • *393*, 394, **424**, 469, 1139
<unique specification> • *570*, 572, **581**, 582, 1229
UNKNOWN • *146*, *153*, 158, *296*
UNNAMED • *145*, *971*, 973, 993, 994, 1214
UNNEST • *146*, 216, 246, 304, 305, 307, *322*
<unqualified schema name> • **159**, 162, 163, 164, 165, 190, 283, 554, 949, 1106, 1107, 1108, 1109
<unquoted date string> • *151*, **152**, 221
<unquoted interval string> • **152**, 219, 221, 225
<unquoted time string> • *151*, **152**, 222, 223, 224, 1232
<unquoted timestamp string> • *151*, **152**, 1232
<unsigned integer> • **151**, *153*, 157, 158, *172*, *173*, 174, 218, 220, *411*, *497*, *498*, *701*, *1041*, 1042, 1043, 1044, 1146, 1221
<unsigned literal> • **150**, *187*
<unsigned numeric literal> • *141*, *150*, **151**
<unsigned value specification> • *185*, 186, **187**, 189, 197, *352*, 353, 356, 357, 358, 359
unterminated C string • 809, 1117
<updatability clause> • **837**, 838, 839, 840, 841, 870, 888, 1022, 1028, 1093, 1153
UPDATE • ?, 99, 118, 119, 120, 133, 134, 135, 548, 549, 550, 551, 567, 568, *583*, 585, 595, 596, 605, *653*, 759, *761*, 762, 763, 770, 778, 781, *837*, 839, 840, 841, *862*, 864, *869*, 870, *871*, 872, 881, 908, 916, *1021*, *1027*, 1028, 1153, 1177, 1200, 1251, 1252
UPDATE • *146*
<update rule> • **583**, 585, 588, 589, 590, 591, 1150
<update source> • 227, 228, 777, 778, 779, 781, 782, 866, 873, **875**, 876, 877, 878, 879, 895, 984, 1153, 1157
<update statement: positioned> • 51, 58, 92, 100, 101, 105, 109, 112, 114, 116, 228, 590, 778, 781, *816*, 856, 866, **869**, 870, 873, 875, 893, 895, 1022, 1027, 1103, 1153, 1228
<update statement: searched> • 51, 58, 105, 109, 112, 113, 115, 228, 595, 777, 778, 779, 781, 782, *816*, **871**, 874, 875, 893, 894, *977*, *1089*, 1103, 1104, 1105, 1153, 1177, 1215, 1227
<update target> • **875**, 877, 878, 879, 984, 1157, 1162
UPPER • 22, *146*, *270*, 280, 412, 415
<upper limit> • **411**, 414
USAGE • ?, ?, ?, ?, ?, 118, 119, *145*, 177, 214, 215, 227, 228, 277, 527, 528, 532, 566, 573, 618, 626, 634, 635, 636, 638, 639, 640, 642, 644, 646, 672, 750, 752, 754, *761*, 762, 763, 772, 775, 776, 777, 778, 779, 780, 781, *1060*, *1061*, 1062, 1063, 1064, 1157, 1203, 1207
USER • *146*, *187*, 188, 191, *560*, 563, *575*, 578, 616, 619, *857*, 858, 860, 863, 868, *935*, 1139, 1140
<user identifier> • 119, *159*, **161**, 167, 935
<user-defined cast definition> • 50, 104, *554*, **729**, 730, *815*, 1103, 1163, 1164
<user-defined character set name> • **527**, 528
<user-defined ordering definition> • 104, 466, 467, 473, *554*, **733**, 735, *815*, 1104, 1168
<user-defined representation> • 47, **659**, 661, 662, 669, 671
<user-defined type body> • **658**, 660
<user-defined type definition> • 43, 44, 47, 53, 86, 104, 177, *554*, **658**, 660, 668, 673, 675, 677, 678, *815*, 1103, 1158, 1236
<user-defined type name> • 40, 41, 82, 87, **161**, 163, 164, 167, *173*, 176, 177, 230, 234, 283, 439, 507, 508, 511, 557, 563, 566, 615, 668, 707, 728, 785, 960, 1032, 1035, 1037
<user-defined type option> • **658**
<user-defined type option list> • **658**, 673, 1158
<user-defined type specification> • **439**
<user-defined type value expression> • **247**, 249, *271*, 282
USER_DEFINED_TYPE_CATALOG • *145*, 958, 960, *971*, 974, 976, 995
USER_DEFINED_TYPE_CODE • *145*, *971*, 973, 995
USER_DEFINED_TYPE_NAME • *145*, 958, 960, *971*, 974, 976, 995
USER_DEFINED_TYPE_SCHEMA • *145*, 958, 960, *971*, 974, 976, 995
USING • *146*, *253*, *254*, 258, 262, *270*, *271*, 273, 278, 281, *330*, *385*, 466, 467, *659*, *862*, *991*, *997*
<using argument> • **997**, 999
<using arguments> • **997**, 999
using clause does not match dynamic parameter specifications • 997, 998, 1117
using clause does not match target specifications • 1002, 1117
using clause required for dynamic parameters • 1007, 1014, 1117
using clause required for result fields • 1007, 1117
<using descriptor> • **991**, *997*
<using input descriptor> • **997**, 998

— V —

VALUE • 52, 69, *146*, *187*, 189, 191, *227*, 296, 297, 574, *581*, 582, *857*, 858, 860, 863, 867, 868, *970*, *973*, 1064, 1138, 1170, 1207
<value expression> • ?, ?, ?, ?, 13, 58, 59, 60, 63, 64, 65, 70, 76, *185*, 186, 197, 202, 205, *208*, *209*, 211, *212*, 213, 214, 215, 216, *230*, **247**, 248, 249, 274, 289, 298, *302*, *308*, *311*, 316, 331, 333, 339, 345, 347, *361*, 362, 363, 364, 365, 366, 369, 374, *385*, 396, 397, 401, 469, 473,

504, 507, 508, 509, *535*, *536*, 537, 538, 539, 542, 543, 544, 545, *546*, *570*, 671, 672, 731, 777, 778, 779, 781, 782, 817, 839, 840, *862*, *875*, 876, 879, 903, 910, *920*, 979, 980, 981, 982, 983, 984, 985, 995, 1109, 1110, 1143, 1144, 1146, 1153, 1172, 1176, 1182, 1202, 1211, 1228, 1235

<value expression primary> • **185**, 186, 197, *229*, *232*, *238*, *241*, *247*, 249, *251*, *265*, *285*, 286, *290*, 291, *300*, *304*, 305, *361*, 362, 363, 601, 610, 619, 620, 622, 754, 755, 775, 776, 777, 778, 779, 781, 981, 1222, 1235, 1236

<value specification> • **187**, 188, 189, 248, 264, 274, 284, 404, 409, 410, 418, 460, 461, 854, 859, 871, 885, *945*, *946*, *948*, *949*, *951*, *952*, *953*, *954*, 984, 986, 1089, 1093, 1139, 1146, 1154, 1155, 1197, 1207, 1236

VALUES • 58, *146*, *316*, 323, 386, 403, 543, *857*, 858, 859, 861, *862*, 1137, 1252

VAR_POP • 64, *146*, *535*, 536, 537, 540, 545, 1189, 1201

VAR_SAMP • 64, *146*, *535*, 536, 537, 540, 545, 1189, 1201

VARBINARY • *146*, *172*, 174, 462, *1046*, 1047, *1053*, 1056, 1058, *1066*, 1068, *1073*, 1074, *1083*, 1085, 1219

VARCHAR • *146*, *171*, 174, 462, *1053*, 1055, 1058, *1072*, 1074, 1075

VARYING • 14, 18, 28, 96, *146*, *171*, *172*, 174, 175, 179, 183, 457, 462, 571, 806, 809, 811, 812, 813, 822, 823, 825, 826, 827, 828, 829, 830, 958, 959, 963, 975, 979, 980, 984, 985, 1005, *1053*, 1055, 1058, 1073, *1083*, 1084, 1085, 1087, 1109

<vertical bar> • 23, *138*, **139**, *411*, 413, 414

VIEW • *145*, 556, 602, 608, 612, *613*, 614, *622*, 623, 624, 651, 732, 737, 784

<view column list> • **613**, 614, 618, 619, 771, 772, 774

<view column option> • **613**, 617

<view definition> • 54, 57, 59, 103, 162, 548, 551, *553*, **613**, 614, 615, 617, 618, 621, 771, 772, 774, *815*, 1104, 1151, 1178, 1233, 1235

<view element> • **613**

<view element list> • **613**, 617

<view specification> • **613**

— W —

warning • 93, 94, 121, 219, 220, 221, 281, 445, 446, 447, 477, 519, 524, 539, 541, 543, 578, 590, 593, 621, 649, 655, 759, 785, 821, 856, 866, 873, 893, 895, 941, 993, 1092, 1106, 1108, 1109, 1113, 1114, 1120

WHEN • *146*, 205, *208*, 209, 210, 304, 305, 307, 387, *653*, *862*, 1110

<when operand> • **208**, 210, 211, 982, 1138

<when operand list> • **208**, 210, 211, 1138

WHENEVER • *146*, *1041*

WHERE • 76, *146*, 240, 244, 323, 334, *339*, 388, *535*, 543, 544, 582, *852*, *854*, 867, *869*, *871*, *1019*, *1021*, *1023*, *1027*, 1104, 1105, 1252

<where clause> • v, 75, 199, 206, *318*, 325, **339**, 341, 347, 354, 364, 368, 373, 1143, 1230, 1233

<white space> • **143**, 146, 147

WHITESPACE • 412, 415

<width bucket bound 1> • **255**, 261

<width bucket bound 2> • **255**, 261

<width bucket count> • **255**, 257, 261

<width bucket function> • 32, *253*, **255**, 257, 261, 264, 1189

<width bucket operand> • **255**, 261

WIDTH_BUCKET • *146*, *255*

WINDOW • *146*, 206, *351*

<window clause> • 62, 200, 202, 206, *318*, 325, **351**, 352, 353, 354, 355, 360, 364, 365, 366, 1182, 1189, 1212

<window definition> • 60, **351**, 352, 353, 354

<window definition list> • **351**

<window frame between> • *351*, **352**, 353

<window frame bound> • **352**

<window frame bound 1> • **352**, 353, 355

<window frame bound 2> • **352**, 353, 355

<window frame clause> • 61, **351**, 353, 355

<window frame exclusion> • 60, *351*, **352**, 359, 360, 1189

<window frame extent> • **351**, 353

<window frame following> • **352**, 353, 356, 358, 359, 985

<window frame preceding> • **352**, 353, 356, 357, 358, 359, 985

<window frame start> • *351*, **352**, 353

<window frame units> • **351**

<window function> • 60, 61, *185*, 186, **204**, 205, 206, 207, 227, 249, 331, 339, 349, 353, 354, 364, 366, 537, 538, 1188

<window function type> • **204**, 206, 366

<window name> • **162**, 168, 169, *204*, 205, 206, 207, *351*, 353, 1188

<window name or specification> • **204**, 206, 207, 1188

<window order clause> • 61, **351**, 352, 353, 355, 358

<window partition clause> • 60, **351**, 352, 353, 354, 469

<window partition column reference> • **351**, 352, 354

<window partition column reference list> • **351**

<window specification> • 60, 61, *204*, 206, 249, **351**, 360, 985, 1188

<window specification details> • 205, **351**

WITH • 10, 15, 33, 34, 35, 36, 37, 59, 74, 92, 98, 100, 118, 119, 120, *146*, 155, 156, 168, *173*, 176, 178, 180, 222, 223, 224, 225, 243, 256, *271*, 285, 288, 293, *322*, 323, 324, 327, 328, 329, 333, 365, 366, *371*, 383, 464, 543, *560*, 562, 568, *613*, 614, 615, 619, 620, 621, *659*, 670, 671, 672, *729*, *733*, *738*, *749*, *751*, 754, 755, 756, 757,

758, 759, 760, 763, *765*, *766*, 769, 772, 773, 774, 775, 776, 778, 779, 782, 783, *835*, 855, 861, 864, 865, 866, 868, 871, 872, 873, 887, 895, 905, 912, *967*, 975, 976, 983, 985, *991*, 992, 993, 1107, 1151, 1161, 1177, 1178, 1184, 1206

with check option violation • 905, 912, 1107, 1121

<with clause> • 77, **371**, 372, 380, 383, 1178

<with column list> • **371**, 372, 380, 385

<with list> • 77, **371**, 372, 380

<with list element> • 55, 57, 325, **371**, 372, 375, 376, 380, 385, 388, 548

<with or without data> • **560**

<with or without time zone> • **173**, 176, 181, 1142, 1232

WITHIN • *146*, *536*

<within group specification> • 202, **536**, 538

WITHOUT • 15, 33, 34, 35, 36, 37, 92, 98, 100, *146*, 155, 156, *173*, 176, 180, 222, 223, 224, 225, 285, 288, 289, 293, *835*, 836, 886, 888, 891, 985, *991*, 1013, 1187

WORK • *145*, *931*, *933*

WRITE • ?, *145*, 924, *925*, 944

— X —

X • *151*

<XQuery option flag> • **253**, 256, 262, 264, *270*, *271*, 273, 274, 278, 281, 284, *417*, 418, 985, 1154, 1155

<XQuery pattern> • **253**, 255, 256, 262, 264, *270*, *271*, 278, 281, 284, *417*, 418, 985, 1154, 1155

<XQuery replacement string> • **271**, 274, 275, 281, 284, 985, 1155, 1198

— Y —

YEAR • 35, 37, *146*, 156, 176, 178, 355, 356, 454, 459, *497*, 498, 976, 980

<year-month literal> • **152**, 156, 499

<years value> • **152**, 156, 499

— Z —

zero-length character string • 191, 268, 284, 1117, 1197, 1198

ZONE • 15, 33, 34, 35, 36, 37, *145*, 155, 156, *173*, 176, 178, 180, 222, 223, 224, 225, 256, *285*, 286, 288, 289, 293, 333, 365, 366, 464, *947*, 975, 976, 983, 985