
**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)*

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

© ISO/IEC 1999

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 734 10 79
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

Contents

Page

Foreword	xv
Introduction	xvii
1 Scope	1
2 Normative references	3
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 Unicode	5
3.1.3 Definitions taken from ISO 8601	6
3.1.4 Definitions taken from Part 1	6
3.1.5 Definitions provided in Part 2	6
3.2 Notation	10
3.3 Conventions	10
3.3.1 Use of terms	10
3.3.1.1 Syntactic containment	10
4 Concepts	11
4.1 Data types	11
4.2 Character strings	13
4.2.1 Character strings and collating sequences	13
4.2.2 Operations involving character strings	14
4.2.2.1 Operators that operate on character strings and return character strings	14
4.2.2.2 Other operators involving character strings	15
4.2.2.3 Operations involving large object character strings	15
4.2.3 Rules determining collating sequence usage	16
4.2.4 Named character sets	18
4.3 Binary strings	20
4.3.1 Binary string comparison	21
4.3.2 Operations involving binary strings	21
4.3.2.1 Operators that operate on binary strings and return binary strings	21
4.3.2.2 Other operators involving binary strings	21
4.4 Bit strings	21
4.4.1 Bit string comparison and assignment	22

4.4.2	Operations involving bit strings	22
4.4.2.1	Operators that operate on bit strings and return bit strings	22
4.4.2.2	Other operators involving bit strings	22
4.5	Numbers	22
4.5.1	Characteristics of numbers	22
4.5.2	Operations involving numbers	23
4.6	Boolean types	24
4.6.1	Comparison and assignment of booleans	24
4.6.2	Operations involving booleans	24
4.6.2.1	Operations on booleans that return booleans	24
4.6.2.2	Other operators involving booleans	24
4.7	Datetimes and intervals	24
4.7.1	Datetimes	25
4.7.2	Intervals	27
4.7.3	Operations involving datetimes and intervals	29
4.8	User-defined types	30
4.8.1	Observers and mutators	32
4.8.2	Constructors	32
4.8.3	Subtypes and supertypes	32
4.8.4	User-defined type comparison and assignment	33
4.8.5	Transforms for user-defined types	34
4.9	Row types	35
4.10	Reference types	35
4.10.1	Operations involving references	36
4.11	Collection types	36
4.11.1	Arrays	36
4.11.2	Collection comparison	37
4.11.3	Operations involving collections	37
4.11.3.1	Operators that operate on array values and return array elements	37
4.11.3.2	Operators that operate on array values and return array values	37
4.12	Type conversions and mixing of data types	37
4.13	Data conversions	39
4.14	Domains	40
4.15	Columns, fields, and attributes	40
4.16	Tables	42
4.16.1	Types of tables	44
4.16.2	Referenceable tables, subtables, and supertables	45
4.16.3	Operations involving tables	46
4.17	Integrity constraints	48
4.17.1	Checking of constraints	48
4.17.2	Table constraints	49
4.17.3	Domain constraints	50
4.17.4	Assertions	50
4.18	Functional dependencies	50
4.18.1	General rules and definitions	51

4.18.2	Known functional dependencies in a base table	52
4.18.3	Known functional dependencies in <table value constructor>	53
4.18.4	Known functional dependencies in a <joined table>	53
4.18.5	Known functional dependencies in a <table reference>	54
4.18.6	Known functional dependencies in the result of a <from clause>	55
4.18.7	Known functional dependencies in the result of a <where clause>	55
4.18.8	Known functional dependencies in the result of a <group by clause>	56
4.18.9	Known functional dependencies in the result of a <having clause>	56
4.18.10	Known functional dependencies in a <query specification>	56
4.18.11	Known functional dependencies in a <query expression>	57
4.19	Candidate keys	58
4.20	SQL-schemas	59
4.21	SQL-client modules	59
4.22	Externally-invoked procedures	60
4.23	SQL-invoked routines	61
4.24	Built-in functions	67
4.25	SQL-paths	67
4.26	Host parameters	68
4.26.1	Status parameters	68
4.26.2	Data parameters	69
4.26.3	Indicator parameters	69
4.26.4	Locators	69
4.27	Diagnostics area	70
4.28	Standard programming languages	70
4.29	Cursors	71
4.30	SQL-statements	73
4.30.1	Classes of SQL-statements	73
4.30.2	SQL-statements classified by function	73
4.30.3	SQL-statements and transaction states	76
4.30.4	SQL-statement atomicity	77
4.31	Basic security model	77
4.31.1	Authorization identifiers	77
4.31.1.1	SQL-session authorization identifiers	78
4.31.1.2	SQL-client module authorization identifiers	79
4.31.1.3	SQL-schema authorization identifiers	79
4.31.2	Privileges	79
4.31.3	Roles	81
4.31.4	Security model definitions	81
4.32	SQL-transactions	82
4.33	SQL-connections	86
4.34	SQL-sessions	87
4.34.1	Execution contexts	89
4.35	Triggers	90
4.35.1	Triggered actions	90
4.35.2	Execution of triggers	91

4.36	Client-server operation	92
5	Lexical elements	93
5.1	<SQL terminal character>	93
5.2	<token> and <separator>	96
5.3	<literal>	105
5.4	Names and identifiers	113
6	Scalar expressions	121
6.1	<data type>	121
6.2	<field definition>	130
6.3	<value specification> and <target specification>	132
6.4	<contextually typed value specification>	136
6.5	<identifier chain>	138
6.6	<column reference>	141
6.7	<SQL parameter reference>	143
6.8	<field reference>	144
6.9	<attribute or method reference>	145
6.10	<method reference>	146
6.11	<method invocation>	147
6.12	<static method invocation>	149
6.13	<element reference>	151
6.14	<dereference operation>	152
6.15	<reference resolution>	153
6.16	<set function specification>	155
6.17	<numeric value function>	159
6.18	<string value function>	164
6.19	<datetime value function>	175
6.20	<interval value function>	177
6.21	<case expression>	178
6.22	<cast specification>	181
6.23	<value expression>	197
6.24	<new specification>	200
6.25	<subtype treatment>	201
6.26	<numeric value expression>	202
6.27	<string value expression>	204
6.28	<datetime value expression>	209
6.29	<interval value expression>	212
6.30	<boolean value expression>	216
6.31	<array value expression>	219
6.32	<array value constructor>	221
7	Query expressions	223
7.1	<row value constructor>	223
7.2	<row value expression>	226
7.3	<table value constructor>	227

7.4	<table expression>	229
7.5	<from clause>	230
7.6	<table reference>	232
7.7	<joined table>	238
7.8	<where clause>	244
7.9	<group by clause>	245
7.10	<having clause>	256
7.11	<query specification>	258
7.12	<query expression>	265
7.13	<search or cycle clause>	279
7.14	<subquery>	283
8	Predicates	285
8.1	<predicate>	285
8.2	<comparison predicate>	287
8.3	<between predicate>	295
8.4	<in predicate>	296
8.5	<like predicate>	298
8.6	<similar predicate>	304
8.7	<null predicate>	309
8.8	<quantified comparison predicate>	310
8.9	<exists predicate>	312
8.10	<unique predicate>	313
8.11	<match predicate>	314
8.12	<overlaps predicate>	316
8.13	<distinct predicate>	318
8.14	<type predicate>	320
8.15	<search condition>	322
9	Data assignment rules and routine determination	323
9.1	Retrieval assignment	323
9.2	Store assignment	328
9.3	Data types of results of aggregations	333
9.4	Subject routine determination	336
9.5	Type precedence list determination	337
9.6	Host parameter mode determination	342
9.7	Type name determination	344
10	Additional common elements	347
10.1	<interval qualifier>	347
10.2	<language clause>	351
10.3	<path specification>	353
10.4	<routine invocation>	354
10.5	<privileges>	374
10.6	<character set specification>	379
10.7	<specific routine designator>	381

10.8	<collate clause>	384
10.9	<constraint name definition> and <constraint characteristics>	385
10.10	Execution of BEFORE triggers	387
10.11	Execution of AFTER triggers	388
10.12	Execution of triggers	389
10.13	Execution of array-returning functions	390
10.14	Data type identity	393
10.15	Determination of a from-sql function	395
10.16	Determination of a from-sql function for an overriding method	396
10.17	Determination of a to-sql function	397
10.18	Determination of a to-sql function for an overriding method	398
11	Schema definition and manipulation	399
11.1	<schema definition>	399
11.2	<drop schema statement>	402
11.3	<table definition>	404
11.4	<column definition>	412
11.5	<default clause>	418
11.6	<table constraint definition>	422
11.7	<unique constraint definition>	424
11.8	<referential constraint definition>	426
11.9	<check constraint definition>	440
11.10	<alter table statement>	442
11.11	<add column definition>	444
11.12	<alter column definition>	445
11.13	<set column default clause>	446
11.14	<drop column default clause>	447
11.15	<add column scope clause>	448
11.16	<drop column scope clause>	449
11.17	<drop column definition>	451
11.18	<add table constraint definition>	453
11.19	<drop table constraint definition>	454
11.20	<drop table statement>	456
11.21	<view definition>	459
11.22	<drop view statement>	469
11.23	<domain definition>	471
11.24	<alter domain statement>	474
11.25	<set domain default clause>	475
11.26	<drop domain default clause>	476
11.27	<add domain constraint definition>	477
11.28	<drop domain constraint definition>	478
11.29	<drop domain statement>	479
11.30	<character set definition>	481
11.31	<drop character set statement>	483
11.32	<collation definition>	485
11.33	<drop collation statement>	487

11.34	<translation definition>	489
11.35	<drop translation statement>	491
11.36	<assertion definition>	493
11.37	<drop assertion statement>	495
11.38	<trigger definition>	497
11.39	<drop trigger statement>	501
11.40	<user-defined type definition>	502
11.41	<attribute definition>	517
11.42	<alter type statement>	520
11.43	<add attribute definition>	521
11.44	<drop attribute definition>	523
11.45	<add original method specification>	525
11.46	<add overriding method specification>	531
11.47	<drop method specification>	535
11.48	<drop data type statement>	537
11.49	<SQL-invoked routine>	541
11.50	<alter routine statement>	562
11.51	<drop routine statement>	565
11.52	<user-defined cast definition>	567
11.53	<drop user-defined cast statement>	569
11.54	<user-defined ordering definition>	571
11.55	<drop user-defined ordering statement>	574
11.56	<transform definition>	576
11.57	<drop transform statement>	579
12	Access control	583
12.1	<grant statement>	583
12.2	<grant privilege statement>	588
12.3	<role definition>	591
12.4	<grant role statement>	592
12.5	<drop role statement>	594
12.6	<revoke statement>	595
13	SQL-client modules	611
13.1	<SQL-client module definition>	611
13.2	<module name clause>	615
13.3	<externally-invoked procedure>	616
13.4	Calls to an <externally-invoked procedure>	619
13.5	<SQL procedure statement>	633
13.6	Data type correspondences	641
14	Data manipulation	651
14.1	<declare cursor>	651
14.2	<open statement>	657
14.3	<fetch statement>	659
14.4	<close statement>	663

14.5	<select statement: single row>	665
14.6	<delete statement: positioned>	667
14.7	<delete statement: searched>	670
14.8	<insert statement>	673
14.9	<update statement: positioned>	677
14.10	<update statement: searched>	684
14.11	<temporary table declaration>	690
14.12	<free locator statement>	692
14.13	<hold locator statement>	693
14.14	Effect of deleting rows from base tables	694
14.15	Effect of deleting some rows from a derived table	696
14.16	Effect of deleting some rows from a viewed table	698
14.17	Effect of inserting tables into base tables	699
14.18	Effect of inserting a table into a derived table	701
14.19	Effect of inserting a table into a viewed table	703
14.20	Effect of replacing rows in base tables	704
14.21	Effect of replacing some rows in a derived table	706
14.22	Effect of replacing some rows in a viewed table	708
15	Control statements	711
15.1	<call statement>	711
15.2	<return statement>	712
16	Transaction management	715
16.1	<start transaction statement>	715
16.2	<set transaction statement>	717
16.3	<set constraints mode statement>	719
16.4	<savepoint statement>	721
16.5	<release savepoint statement>	722
16.6	<commit statement>	723
16.7	<rollback statement>	725
17	Connection management	727
17.1	<connect statement>	727
17.2	<set connection statement>	730
17.3	<disconnect statement>	732
18	Session management	735
18.1	<set session characteristics statement>	735
18.2	<set session user identifier statement>	736
18.3	<set role statement>	737
18.4	<set local time zone statement>	738
19	Diagnostics management	739
19.1	<get diagnostics statement>	739

20	Information Schema	751
20.1	Introduction to Information Schema and Definition Schema	751
20.2	INFORMATION_SCHEMA Schema	752
20.3	INFORMATION_SCHEMA_CATALOG_NAME base table	753
20.4	CARDINAL_NUMBER domain	754
20.5	CHARACTER_DATA domain	754
20.6	SQL_IDENTIFIER domain	755
20.7	TIME_STAMP domain	755
20.8	ADMINISTRABLE_ROLE_AUTHORIZATIONS view	756
20.9	APPLICABLE_ROLES view	757
20.10	ASSERTIONS view	758
20.11	ATTRIBUTES view	759
20.12	CHARACTER_SETS view	761
20.13	CHECK_CONSTRAINTS view	762
20.14	COLLATIONS view	763
20.15	COLUMN_DOMAIN_USAGE view	764
20.16	COLUMN_PRIVILEGES view	765
20.17	COLUMN_UDT_USAGE view	766
20.18	COLUMNS view	767
20.19	CONSTRAINT_COLUMN_USAGE view	769
20.20	CONSTRAINT_TABLE_USAGE view	770
20.21	DATA_TYPE_PRIVILEGES view	771
20.22	DIRECT_SUPERTABLES view	772
20.23	DIRECT_SUPERTYPES view	773
20.24	DOMAIN_CONSTRAINTS view	774
20.25	DOMAIN_UDT_USAGE view	775
20.26	DOMAINS view	776
20.27	ELEMENT_TYPES view	777
20.28	ENABLED_ROLES view	778
20.29	FIELDS view	779
20.30	KEY_COLUMN_USAGE view	780
20.31	METHOD_SPECIFICATION_PARAMETERS view	781
20.32	METHOD_SPECIFICATIONS view	783
20.33	PARAMETERS view	785
20.34	REFERENCED_TYPES view	787
20.35	REFERENTIAL_CONSTRAINTS view	788
20.36	ROLE_COLUMN_GRANTS view	789
20.37	ROLE_ROUTINE_GRANTS view	790
20.38	ROLE_TABLE_GRANTS view	791
20.39	ROLE_TABLE_METHOD_GRANTS view	792
20.40	ROLE_USAGE_GRANTS view	793
20.41	ROLE_UDT_GRANTS view	794
20.42	ROUTINE_COLUMN_USAGE view	795
20.43	ROUTINE_PRIVILEGES view	796
20.44	ROUTINE_TABLE_USAGE view	797

20.45	ROUTINES view	798
20.46	SCHEMATA view	800
20.47	SQL_FEATURES view	801
20.48	SQL_IMPLEMENTATION_INFO view	802
20.49	SQL_LANGUAGES view	803
20.50	SQL_PACKAGES view	804
20.51	SQL_SIZING view	805
20.52	SQL_SIZING_PROFILES view	806
20.53	TABLE_CONSTRAINTS view	807
20.54	TABLE_METHOD_PRIVILEGES view	808
20.55	TABLE_PRIVILEGES view	809
20.56	TABLES view	810
20.57	TRANSFORMS view	811
20.58	TRANSLATIONS view	812
20.59	TRIGGERED_UPDATE_COLUMNS view	813
20.60	TRIGGER_COLUMN_USAGE view	814
20.61	TRIGGER_TABLE_USAGE view	815
20.62	TRIGGERS view	816
20.63	USAGE_PRIVILEGES view	817
20.64	UDT_PRIVILEGES view	818
20.65	USER_DEFINED_TYPES view	819
20.66	VIEW_COLUMN_USAGE view	821
20.67	VIEW_TABLE_USAGE view	822
20.68	VIEWS view	823
20.69	Short name views	824
20.70	Definition of SQL built-in functions	835
21	Definition Schema	847
21.1	Introduction to the Definition Schema	847
21.2	DEFINITION_SCHEMA Schema	848
21.3	EQUAL_KEY_DEGREES assertion	849
21.4	KEY_DEGREE_GREATER_THAN_OR_EQUAL_TO_1 assertion	850
21.5	UNIQUE_CONSTRAINT_NAME assertion	851
21.6	ASSERTIONS base table	852
21.7	ATTRIBUTES base table	853
21.8	CHARACTER_SETS base table	855
21.9	CHECK_COLUMN_USAGE base table	857
21.10	CHECK_TABLE_USAGE base table	858
21.11	CHECK_CONSTRAINTS base table	859
21.12	COLLATIONS base table	860
21.13	COLUMN_PRIVILEGES base table	862
21.14	COLUMNS base table	864
21.15	DATA_TYPE_DESCRIPTOR base table	867
21.16	DIRECT_SUPERTABLES base table	874
21.17	DIRECT_SUPERTYPES base table	876
21.18	DOMAIN_CONSTRAINTS base table	878

21.19	DOMAINS base table	880
21.20	ELEMENT_TYPES base table	881
21.21	FIELDS base table	882
21.22	KEY_COLUMN_USAGE base table	884
21.23	METHOD_SPECIFICATION_PARAMETERS base table	886
21.24	METHOD_SPECIFICATIONS base table	888
21.25	PARAMETERS base table	891
21.26	REFERENCED_TYPES base table	893
21.27	REFERENTIAL_CONSTRAINTS base table	894
21.28	ROLE_AUTHORIZATION_DESCRIPTOR base table	896
21.29	ROLES base table	898
21.30	ROUTINE_COLUMN_USAGE base table	899
21.31	ROUTINE_PRIVILEGES base table	901
21.32	ROUTINE_TABLE_USAGE base table	903
21.33	ROUTINES base table	905
21.34	SCHEMATA base table	910
21.35	SQL_FEATURES base table	911
21.36	SQL_IMPLEMENTATION_INFO base table	913
21.37	SQL_LANGUAGES base table	914
21.38	SQL_SIZING base table	918
21.39	SQL_SIZING_PROFILES base table	919
21.40	TABLE_CONSTRAINTS base table	920
21.41	TABLE_METHOD_PRIVILEGES base table	922
21.42	TABLE_PRIVILEGES base table	924
21.43	TABLES base table	926
21.44	TRANSFORMS base table	928
21.45	TRANSLATIONS base table	929
21.46	TRIGGERED_UPDATE_COLUMNS base table	931
21.47	TRIGGER_COLUMN_USAGE base table	932
21.48	TRIGGER_TABLE_USAGE base table	934
21.49	TRIGGERS base table	936
21.50	USAGE_PRIVILEGES base table	938
21.51	USER_DEFINED_TYPE_PRIVILEGES base table	940
21.52	USER_DEFINED_TYPES base table	942
21.53	USERS base table	945
21.54	VIEW_COLUMN_USAGE base table	946
21.55	VIEW_TABLE_USAGE base table	947
21.56	VIEWS base table	948
22	Status codes	951
22.1	SQLSTATE	951
22.2	Remote Database Access SQLSTATE Subclasses	958
22.3	SQL Multimedia and Application Packages SQLSTATE Subclasses	958

23	Conformance	961
23.1	General conformance requirements	961
23.2	Claims of conformance	962
Annex A	SQL Conformance Summary	965
Annex B	Implementation-defined elements	1017
Annex C	Implementation-dependent elements	1027
Annex D	Deprecated features	1033
Annex E	Incompatibilities with ISO/IEC 9075:1992 and ISO/IEC 9075-4:1996	1035
Annex F	SQL feature and package taxonomy	1041
Index		1065

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

TABLES

Tables	Page
1 Collating coercibility rules for monadic operators	17
2 Collating coercibility rules for dyadic operators	17
3 Collating sequence usage for comparisons	18
4 Fields in datetime values	25
5 Datetime data type conversions	27
6 Fields in year-month INTERVAL values	27
7 Fields in day-time INTERVAL values	28
8 Valid values for fields in INTERVAL values	28
9 Valid operators involving datetimes and intervals	29
10 SQL-transaction isolation levels and the three phenomena	84
11 Valid values for datetime fields	127
12 Valid absolute values for interval fields	128
13 Truth table for the AND boolean operator	218
14 Truth table for the OR boolean operator	218
15 Truth table for the IS boolean operator	218
16 <null predicate> semantics	309
17 Standard programming languages	352
18 Data type correspondences for Ada	641
19 Data type correspondences for C	642
20 Data type correspondences for COBOL	644
21 Data type correspondences for Fortran	646
22 Data type correspondences for MUMPS	647
23 Data type correspondences for Pascal	648
24 Data type correspondences for PL/I	649
25 <identifier>s for use with <get diagnostics statement>	741
26 SQL-statement codes	742
27 SQLSTATE class and subclass values	952
28 SQLSTATE class codes for RDA	958
29 SQLSTATE class codes for SQL/MM	959
30 Implied feature relationships	961
31 SQL/Foundation feature taxonomy and definition for Core SQL	1042
32 SQL/Foundation feature taxonomy for features outside Core SQL	1057

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.

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

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 fourth edition cancels and replaces the third edition (ISO/IEC 9075:1992).

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 5: Host Language Bindings (SQL/Bindings)*

Annexes A, B, C, D, E, and F of this part of ISO/IEC 9075 are for information only.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

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, "Data assignment rules and routine determination", 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, "Control statements", defines the SQL-control statements.
- 16) Clause 16, "Transaction management", defines the SQL-transaction management statements.
- 17) Clause 17, "Connection management" defines the SQL-connection management statements.
- 18) Clause 18, "Session management", defines the SQL-session management statements.
- 19) Clause 19, "Diagnostics management", defines the diagnostics management facilities.
- 20) Clause 20, "Information Schema", defines viewed tables that contain schema information.
- 21) Clause 21, "Definition Schema", defines base tables on which the viewed tables containing schema information depend.

- 22) Clause 22, "Status codes", defines values that identify the status of the execution of SQL-statements and the mechanisms by which those values are returned.
- 23) Clause 23, "Conformance", defines the criteria for conformance to this part of ISO/IEC 9075.
- 24) Annex A, "SQL Conformance Summary", is an informative Annex. It summarizes the conformance requirements of the SQL language.
- 25) 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.
- 26) 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.
- 27) 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.
- 28) Annex E, "Incompatibilities with ISO/IEC 9075:1992 and ISO/IEC 9075-4:1996", is an informative Annex. It lists incompatibilities with the previous version of this part of ISO/IEC 9075.
- 29) Annex F, "SQL feature and package taxonomy", is an informative Annex. It identifies features and packages 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 both to Core SQL and to the packages specified in this part of ISO/IEC 9075. The feature taxonomy may be used to develop other profiles involving the SQL language.

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 22, "Status codes", Subclauses begin a new page. Any resulting blank space is not significant.

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.

It also specifies an Information Schema that describes the structure and the integrity constraints of SQL-data.

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.

This part of ISO/IEC 9075 does not define the method or time of binding between any of:

- database management system components,
- SQL data definition declarations,
- SQL procedures, or
- compilation units.

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.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

2 Normative references

The following standards contain provisions that, through reference in this text, constitute provisions of this part of ISO/IEC 9075. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 9075 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid international Standards.

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

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

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

ISO/IEC 2022:1994, *Information technology — Character code structure and extension techniques*.

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

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

ISO 8601:1988, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

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

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

ISO 8824-1:1995, *Information technology — Specification of Abstract Syntax Notation One (ASN.1) — Part 1: Specification of basic notation*.

ISO/IEC 9075-1:1999, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*.

ISO/IEC FDIS 9075-3:1999, *Information technology — Database languages — SQL — Part 3: Call-Level Interface (SQL/CLI)*.

ISO/IEC 9075-4:1999, *Information technology — Database languages — SQL — Part 4: Persistent Stored Modules (SQL/PSM)*.

ISO/IEC 9075-5:1999, *Information technology — Database languages — SQL — Part 5: Host Language Bindings (SQL/Bindings)*.

ISO/IEC 9579-1:1993, *Information technology — Open Systems Interconnection — Remote Database Access — Part 1: Generic Model, Service, and Protocol.*

ISO/IEC 9579-2:1998, *Information technology — Open Systems Interconnection — Remote Database Access — Part 2: SQL Specialization.*

ISO/IEC 9899:1990, *Programming languages — C.*

ISO/IEC 9899:1990/Amendment 1:1995, *Amendment to ISO/IEC 9899:1990 — C Integrity.*

ISO/IEC 10026-2:1996, *Information technology — Open Systems Interconnection — Distributed Transaction Processing — Part 2: OSI TP Service.*

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

ISO/IEC 11404:1996, *Information technology — Programming languages, their environments and system software interfaces — Language-independent datatypes.*

ISO/IEC 11756:1992, *Information technology — Programming languages — MUMPS.*

ISO/IEC 10646-1:1993, *Information technology — Universal Multi-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

The Unicode Consortium, *The Unicode Standard, Version 2.0*, 1996. ISBN 0-201-48345-9.

3 Definitions, notations, and conventions

3.1 Definitions

For the purposes of this part of ISO/IEC 9075, the following definitions apply.

3.1.1 Definitions taken from ISO/IEC 10646

This part of ISO/IEC 9075 makes use of the following terms defined in ISO/IEC 10646:

- a) **character**
- b) **coded character**
- c) **coded character set**
- d) **control function**
- e) **private use plane**
- f) **repertoire**

3.1.2 Definitions taken from Unicode

This part of ISO/IEC 9075 makes use of the following terms defined in The Unicode Standard:

- a) **abstract character**
- b) **abstract character sequence**
- c) **canonical decomposition**
- d) **canonical equivalent**
- e) **code value**
- f) **compatibility decomposition**
- g) **compatibility equivalent**
- h) **control character**
- i) **private use**
- j) **surrogate pair**

3.1 Definitions

3.1.3 Definitions taken from ISO 8601

This part of ISO/IEC 9075 makes use of the following terms defined in ISO 8601:

- a) **Coördinated Universal Time** (UTC)
- b) **date** (“date, calendar” in ISO 8601)

3.1.4 Definitions taken from Part 1

This part of ISO/IEC 9075 makes use of all terms defined in ISO/IEC 9075-1.

3.1.5 Definitions provided in Part 2

This part of ISO/IEC 9075 defines the following terms:

- a) **assignable**: The characteristic of a data type that permits a value of that data type to be assigned to a site of a specified data type. See Subclause 4.12, “Type conversions and mixing of data types”.
- b) **assignment**: An operation that replaces the instance at a site (known as the *target*) with a new instance of a (possibly, though not necessarily, different) value (known as the *source*).
- c) **attribute**: A component (of a structured type) whose characteristics are specified by an attribute descriptor. A value *V* in structured type *T* has exactly one attribute value for each attribute *A* of *T*, this being the result of the invocation of *A(V)* of the observer function for that attribute.
- d) **cardinality** (of a value of a collection type): The number of elements in that value. Those elements need not necessarily have distinct values.
- e) **character repertoire, repertoire**: A set of characters used for a specific purpose or application. Each character repertoire has an implied default collating sequence.
- f) **coercibility**: A characteristic of a character string value that governs how a collating sequence for the value is determined.
- g) **collation, collating sequence**: A method of ordering two comparable character strings. Every character set has a default collation.
- h) **collection type**: The type of collection denoted by a collection data type: array.
- i) **comparable**: The characteristic of values that permits one value to be compared with another value. Also said of data types: Two data types are comparable if values of those data types are comparable. If one of the two data types is a user-defined type, then both shall be in the same subtype family. See Subclause 4.12, “Type conversions and mixing of data types”.
- j) **constructor function**: A niladic SQL-invoked function of which exactly one is implicitly specified for every structured type. An invocation of the constructor function for data type *T* returns a value *V* of the most specific type *T* such that *V* is not null and, for every observer function *O* defined for *T*, the invocation *O(V)* returns the default value of the attribute corresponding to *O*.

- k) **declared type:** Of an expression denoting a value, the unique data type that is common to every value that might result from evaluation of that expression. The term is also applicable to anything that can be referenced to denote a value, such as, for example, a parameter, column or variable.
- l) **distinct:** Two values other than rows and collections are said to be *not distinct* if either: both are the null value, or they compare equal according to Subclause 8.2, "<comparison predicate>"; otherwise they are distinct. Two rows (or partial rows) are distinct if at least one of their pairs of respective values is distinct; otherwise they are not distinct. Two arrays are distinct if the arrays do not have the same cardinality or if, for arrays of the same cardinality, elements in the same ordinal position in the two arrays are distinct; otherwise, the arrays are not distinct. The result of evaluating whether or not two values, two rows, or two arrays are distinct is never unknown.
- m) **duplicate:** Two or more values or rows are said to be duplicates (of each other) if and only if they are not distinct.
- n) **dyadic:** Of operators, functions, and procedures, having exactly two operands or parameters. 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.
- o) **element type:** The definition of a collection type *CT* includes specification of a data type that is common to every element of every value in *CT*. This is the element type of *CT* and also the element type of every value in *CT*.
- p) **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.
- q) **fixed-length:** A characteristic of bit strings and character strings that restricts a string to contain exactly one number of bits or characters, respectively, known as the length in bits or characters, respectively, of the string.
- r) **form-of-use:** A convention (or encoding) for representing characters (in character strings). Some forms-of-use are fixed-length codings and others are variable-length codings.
- s) **form-of-use conversion:** A method of converting character strings from one form-of-use to another form-of-use.
- t) **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.
- u) **monadic:** Of operators, functions, and procedures, having exactly one operand or parameter. 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.
- v) **most specific type:** Of a value, the unique data type of which every data type of that value is a supertype.
- w) **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*. 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 Definitions

- x) ***n*-adic operator**: An operator having a variable number of operands (informally: *n* operands). An example of an *n*-adic operator in this part of ISO/IEC 9075 is COALESCE.
- y) **niladic**: Of functions and procedures, having no parameters.
- z) **observer function**: An SQL-invoked function *A* 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 *A*(*V*) returns some value *AV* in *AT*. *AV* is then said to be the value of attribute *A* in *V*.
- aa) **redundant duplicates**: All except one of any multiset of duplicate values or rows.
- bb) **reference type**: A data type all of whose values are potential references to sites of one specified data type.
- cc) **REF value**: A value that references some site.
- dd) **referenced type**: The declared type of the values at sites referenced by values of a particular reference type.
- ee) **referenced value**: The value at the site referenced by a REF value.
- ff) **repertoire**: See **character repertoire**.
- gg) **result SQL parameter**: An SQL parameter that specifies RESULT.
- hh) **result data type**: The result data type of an SQL-invoked function is the declared type of the result of its invocation.
- ii) **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.
- jj) **SQL argument**: An expression denoting a value to be substituted for an SQL parameter in an invocation of an SQL-invoked routine.
- kk) **SQL-invoked routine**: A routine that is allowed to be invoked only from within SQL.
- ll) **SQL parameter**: A parameter declared as part of the signature of an SQL-invoked routine.
- mm) **SQL routine**: An SQL-invoked routine whose routine body is written in SQL.
- nn) **STATE function**: A dyadic BOOLEAN function, with <schema qualified routine name> EQUALS, which can be implicitly defined in connection with a structured type whose values are unordered. The parameters of the STATE function for data type *T* are both of declared type *T*. The invocation EQUALS(*V1*, *V2*) returns *true* if and only if, for every attribute *A* of *T*, *A*(*V1*) = *A*(*V2*).
- oo) **subfield**: A field *F* is a subfield of a row type *RT* if *F* is a field of *RT* or *F* is a field of a row type *RT2* that is the declared type of a field *F2* that is a subfield of *RT*.
- pp) **subtype**: A data type *T2* is a subtype of data type *T1* if every value of *T2* is also a value of *T1*. If *T1* and *T2* are not compatible, then *T2* is a *proper subtype* of *T1*. "Compatible" is defined in Subclause 4.12, "Type conversions and mixing of data types". See also **supertype**.

- qq) **supertype**: A data type *T1* is a supertype of data type *T2* if every value of *T2* is also a value of *T1*. If *T1* and *T2* are not compatible, then *T2* is a *proper supertype* of *T1*. "Compatible" is defined in Subclause 4.12, "Type conversions and mixing of data types". See also **subtype**.
- rr) **translation**: A method of translating characters in one character repertoire into characters of the same or a different character repertoire.
- ss) **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.
- tt) **user-defined type**: A type whose characteristics are specified by a user-defined type descriptor.
- uu) **variable-length**: A characteristic of bit strings and character strings that allows a string to contain any number of bits or characters, respectively, between 0 (zero) and some maximum number, known as the maximum length in bits or characters, respectively, of the string.
- vv) **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 of the following characters:
- U+0009, Horizontal Tab
 - U+000A, Line Feed
 - U+000B, Vertical Tabulation
 - U+000C, Form Feed
 - U+000D, Carriage Return
 - U+0020, Space
 - U+00A0, No-Break Space
 - 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

3.1 Definitions

- U+200A, Hair Space
- U+200B, Zero Width Space
- U+200C, Zero Width Non-Joiner
- U+200D, Zero Width Joiner
- U+200E, Left-To-Right Mark
- U+200F, Right-To-Left Mark
- U+3000, Ideographic Space
- U+2028, Line Separator
- U+2029, Paragraph Separator
- U+FEFF, Zero Width No-Break Space

NOTE 1 – The notation “U+xyzw” identifies a character position on the Basic Multilingual Plane of ISO/IEC 10646, where each of x, y, z, and w are hexadecimal digits; that character position is in column zw of row xy. In this International Standard, this notation is used only to unambiguously identify characters and is not meant to imply a specific encoding for any implementation’s use of that character.

3.2 Notation

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

3.3 Conventions

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

3.3.1 Use of terms

3.3.1.1 Syntactic containment

Let <A> and be syntactic elements; let *A1* be an instance of <A> and let *B1* be an instance of .

A1 *directly contains* *B1* if *A1* contains *B1* without an intervening <set function specification> or <subquery>.

4 Concepts

4.1 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 the “built-in data types”, though not in this International Standard. User-defined data 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, REF, and ROW. A constructed type is either an array type, a reference type or a row type, according to whether it is specified with ARRAY, REF, or ROW, respectively. Array types are the only examples of constructed types 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 4.7, “Datetimes and intervals”.

Row types, reference types and collection types are described in Subclause 4.9, “Row types”, Subclause 4.10, “Reference types”, Subclause 4.11, “Collection types”, respectively.

SQL defines predefined data types named by the following <key word>s: CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT, BINARY LARGE OBJECT, BIT, BIT VARYING, NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE PRECISION, BOOLEAN, DATE, TIME, TIMESTAMP, and INTERVAL. These names are used in the *type designators* that constitute the *type precedence lists* specified in Subclause 9.5, “Type precedence list determination”. In fact, every data type is assigned to exactly one equivalence class named by such a type designator.

For reference purposes:

- The data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are collectively referred to as *character string types*.
- The data types BIT and BIT VARYING are collectively referred to as *bit string types*.
- The data type BINARY LARGE OBJECT is referred to as the *binary string type* and the values of binary string types are referred to as *binary strings*.
- The data types CHARACTER LARGE OBJECT and BINARY LARGE OBJECT are collectively referred to as *large object string types* and the values of large object string types are referred to as *large object strings*.

4.1 Data types

- Character string types, bit 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, INTEGER and SMALLINT 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 an instance of that data type.

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

A structured type *ST* is *directly based on* a data type *DT* if *DT* is the declared type of some attribute whose descriptor is included in the descriptor of *ST*.

An array type *AT* is *directly based on* a data type *DT* if *DT* is the element type of *AT*.

A reference type *RT* is *directly based on* a data type *DT* if *DT* is the referenced type.

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 directly based on *DT2* or *DT1* is directly based on some data type that is based on *DT2*.

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 11.49, "<SQL-invoked routine>", and Subclause 16.1, "<embedded SQL host program>", in ISO/IEC 9075-5. Not every SQL data type has a corresponding data type in every host language.

4.2 Character strings

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

- The name of the specific character string data 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 data type.
- The catalog name, schema name, and character set name of the character set of the character string data type.
- The catalog name, schema name, and collation name of the collation of the character string data type.

Character sets fall into three categories: those defined by national or international standards, those defined by implementations, and those defined by applications. 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 implementations reside in the Information Schema (named INFORMATION_SCHEMA) in each catalog, as do collations defined by standards and collations, translations, and form-of-use conversions defined by implementations.

4.2.1 Character strings and collating sequences

A character string is a sequence of characters chosen from the same character repertoire. The character repertoire from which the characters of a particular string are chosen may be specified explicitly or implicitly. A character string has a length, which is the number of characters in the sequence. The length is 0 (zero) or a positive integer.

All character strings of a given character repertoire are comparable.

A collating sequence, also known as a collation, is a set of rules determining comparison of character strings in a particular character repertoire. There is a default collating sequence for each character repertoire, but additional collating sequences can be defined for any character repertoire.

NOTE 2 – A column may be defined as having a default collating sequence. This default collating sequence for the column may be different from the default collating sequence for its character repertoire, *e.g.*, if the <collate clause> is specified in the column reference. It will be clear from context when the term “default collating sequence” is used whether it is meant for a column or for a character repertoire.

Given a collating sequence, two character strings are identical if and only if they are equal in accordance with the comparison rules specified in Subclause 8.2, “<comparison predicate>”. The collating sequence used for a particular comparison is determined as in Subclause 4.2.3, “Rules determining collating sequence usage”.

The <key word>s NATIONAL CHARACTER are used to specify a character string data type with a particular implementation-defined character repertoire. Special syntax (N’string’) is provided for representing literals in that character repertoire.

4.2 Character strings

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 default collation for the character set.

For every character set, there is at least one collation. A collation is described by a collation descriptor. A collation descriptor includes:

- The name of the collation.
- The name of the character set on which the collation operates.
- Whether the collation has the NO PAD or the PAD SPACE characteristic.

4.2.2 Operations involving character strings

4.2.2.1 Operators that operate on character strings and return character strings

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

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

<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 that has the Unicode 'alphabetic' property and whose Unicode name includes 'lower'. An upper case character is a character that has the Unicode 'alphabetic' property and whose Unicode name includes 'upper'. A title case character is a character that has the Unicode 'alphabetic' property and whose Unicode name includes 'title'.

<form-of-use conversion> is a function that invokes an installation-supplied form-of-use conversion 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 some different form-of-use. 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 translation> 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 <translation name>.

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

- The name of the translation,.
- The name of the character set from which it translates.
- The name of the character set to which it translates.
- An indication of how the translation is performed.

4.2.2.2 Other operators involving character strings

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

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

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

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

4.2.2.3 Operations involving large object character strings

Large object strings cannot be operated on by all string operations. Large object 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>.

4.2 Character strings

As a result of these restrictions, large object strings and large object string columns cannot be referenced 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 4.2.2.1, “Operators that operate on character strings and return character strings”, and Subclause 4.2.2.2, “Other operators involving character strings”, are supported for large object character strings.

4.2.3 Rules determining collating sequence usage

The rules determining collating sequence usage for character strings are based on the following:

- Expressions where no columns are involved (e.g., literals, host variables) are by default compared using the default collating sequence for their character repertoire.
NOTE 3 – The default collating sequence for a character repertoire is defined in Subclause 10.6, “<character set specification>”, and Subclause 11.30, “<character set definition>”.
- When one or more columns are involved (e.g., comparing two columns, or comparing a column to a literal), then provided that all columns involved have the same default collating sequence and there is no explicit specification of a collating sequence, that default collating sequence is used.
- When columns are involved having different default collating sequences, explicit specification of the collating sequence in the expression is required via the <collate clause>.
- Any explicit specification of collating sequence in an expression overrides any default collating sequence.

To formalize this, <character value expression>s effectively have a coercibility characteristic. This characteristic has the values *Coercible*, *Implicit*, *No collating sequence*, and *Explicit*. <character value expression>s with the *Coercible*, *Implicit*, or *Explicit* characteristics have a collating sequence.

A <character value expression> consisting of a column reference has the coercibility characteristic *Implicit*, with collating sequence as defined when the column was created. A <character value expression> consisting of a value other than a column (e.g., a host variable or a literal) has the coercibility characteristic *Coercible*, with the default collation for its character repertoire. A <character value expression> simply containing a <collate clause> has the coercibility characteristic *Explicit*, with the collating sequence specified in the <collate clause>.

NOTE 4 – When the coercibility characteristic is *Coercible*, the collating sequence is uniquely determined as specified in Subclause 8.2, “<comparison predicate>”.

The tables below define how the collating sequence and the coercibility characteristic is determined for the result of any monadic or dyadic operation. Table 1, “Collating coercibility rules for monadic operators”, shows the collating sequence and coercibility rules for monadic operators, and Table 2, “Collating coercibility rules for dyadic operators”, shows the collating sequence and coercibility rules for dyadic operators. Table 3, “Collating sequence usage for comparisons”, shows how the collating sequence is determined for a particular comparison.

Table 1—Collating coercibility rules for monadic operators

Operand Coercibility and Collating Sequence		Result Coercibility and Collating Sequence	
Coercibility	Collating Sequence	Coercibility	Collating Sequence
Coercible	default	Coercible	default
Implicit	X	Implicit	X
Explicit	X	Explicit	X
No collating sequence		No collating sequence	

Table 2—Collating coercibility rules for dyadic operators

Operand 1 Coercibility and Collating Sequence		Operand 2 Coercibility and Collating Sequence		Result Coercibility and Collating Sequence	
Coercibility	Collating Sequence	Coercibility	Collating Sequence	Coercibility	Collating Sequence
Coercible	default	Coercible	default	Coercible	default
Coercible	default	Implicit	Y	Implicit	Y
Coercible	default	No collating sequence		No collating sequence	
Coercible	default	Explicit	Y	Explicit	Y
Implicit	X	Coercible	default	Implicit	X
Implicit	X	Implicit	X	Implicit	X
Implicit	X	Implicit	$Y \neq X$	No collating sequence	
Implicit	X	No collating sequence		No collating sequence	
No collating sequence		Explicit	Y	Explicit	Y
No collating sequence		Any, except Explicit	Any	No collating sequence	
No collating sequence		Explicit	X	Explicit	X
Explicit	X	Coercible	default	Explicit	X
Explicit	X	Implicit	Y	Explicit	X
Explicit	X	No collating sequence		Explicit	X
Explicit	X	Explicit	X	Explicit	X
Explicit	X	Explicit	$Y \neq X$	Not permitted: <i>invalid syntax</i>	

4.2 Character strings

Table 3—Collating sequence usage for comparisons

Comparand 1 Coercibility and Collating Sequence		Comparand 2 Coercibility and Collating Sequence		Collating Sequence Used For The Comparison
Coercibility	Collating Sequence	Coercibility	Collating Sequence	
Coercible	default	Coercible	default	default
Coercible	default	Implicit	Y	Y
Coercible	default	No collating sequence		Not permitted: <i>invalid syntax</i>
Coercible	default	Explicit	Y	Y
Implicit	X	Coercible	default	X
Implicit	X	Implicit	X	X
Implicit	X	Implicit	$Y \neq X$	Not permitted: <i>invalid syntax</i>
Implicit	X	No collating sequence		Not permitted: <i>invalid syntax</i>
Implicit	X	Explicit	Y	Y
No collating sequence		Any except Explicit	Any	Not permitted: <i>invalid syntax</i>
No collating sequence		Explicit	X	X
Explicit	X	Coercible	default	X
Explicit	X	Implicit	Y	X
Explicit	X	No collating sequence		X
Explicit	X	Explicit	X	X
Explicit	X	Explicit	$Y \neq X$	Not permitted: <i>invalid syntax</i>

For n -adic operations (e.g., <case expression>) with operands $X_1, X_2, \dots, X_n$, the collating sequence is effectively determined by considering X_1 and X_2 , then combining this result with X_3 , and so on.

4.2.4 Named character sets

An SQL <character set specification> allows a reference to a character set name defined by a standard, an implementation, or a user. The following SQL supported character set names are specified as part of ISO/IEC 9075:

- SQL_CHARACTER specifies the name of a character repertoire that consists of the 88 <SQL language character>s as specified in Subclause 5.1, “<SQL terminal character>”. It consists of the 52 uppercase and lowercase simple latin characters, 10 digits, and 26 <SQL special character>s, including: <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>, <underscore>, <vertical bar>, <left bracket>, <right bracket>, <circumflex>, <left brace>, and <right brace>. The 88 characters specified as <SQL language character>s are all included in the ISO International Reference Version (IRV) characters specified in ISO 646:1991. The characters in IRV are included in many other international character set definitions. In addition, 82 of these characters (all except <vertical bar>, <left bracket>, <right bracket>, <circumflex>, <left brace>, and <right brace>) are in the most stable subset of IRV that, by ISO

convention, is included in every latin-based ISO standard set of characters. As far as can be determined, <vertical bar>, <left bracket>, <right bracket>, <circumflex>, <left brace>, and <right brace> are included in most character sets that enjoy world-wide use. Thus, the SQL_CHARACTER repertoire is the most universal of the character sets named herein. The collation and form-of-use of SQL_CHARACTER is implementation-defined.

- GRAPHIC_IRV (or ASCII_GRAPHIC) specifies the name of a character repertoire that consists of the 95-character graphic subset of the International Reference Version (IRV) as specified in ISO 646:1991. The form-of-use is that corresponding to the coded representation of each character by a single byte (possibly 7-bit, 8-bit, or other), with no designation escape sequences for other character sets. The default collating sequence is that corresponding to the bit combinations defined by ISO 646:1991. The GRAPHIC_IRV character set is a superset of the <SQL language character>s. The 7 characters included in GRAPHIC_IRV that are not <SQL language character>s are, in collation order: Exclamation mark (!), Number sign (#), Dollar sign (\$), Commercial at (@), Reverse solidus (\), Grave accent (`), and Tilde (~). Of these 7 characters, only “!” is in the most stable subset of ISO 646, whereas “#” competes with the British pound sign, “\$” competes with the international currency symbol, and the others occupy positions in ISO 646 that are reserved for national or application-oriented use. However, all are the default IRV values specified in ISO 646 when no national or application-specific version is explicitly specified.
- LATIN1 specifies the name of a character repertoire that consists of the 191 graphic characters defined in ISO 8859-1. The form-of-use is that corresponding to the coded representation of each character by a single 8-bit byte, with no designation escape sequences for other character sets. The default collating sequence is that corresponding to the bit combinations defined by ISO 8859-1. The LATIN1 character set is a superset of GRAPHIC_IRV and, when restricted to the GRAPHIC_IRV characters, produces the same collation as GRAPHIC_IRV. LATIN1 consists of all characters commonly used in the following languages: Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, and Swedish. It also includes the following special symbols, in collation order: No-break space, Inverted exclamation mark (¡), Cent sign (¢), Pound sign (£), Currency sign (₹), Yen sign (¥), Broken bar (|), Paragraph sign (¶), Diaeresis (¨), Copyright sign (©), Feminine ordinal indicator (ª), Left angle quotation mark («), Not sign (¬), Soft hyphen, Registered trade mark sign (®), Macron (ˆ), Degree sign (°), Plus-minus sign (±), Superscript two (²), Superscript three (³), Acute accent (´), Micro sign (µ), Pilcrow sign, Middle dot (·), Cedilla (¸), Superscript one (¹), Masculine ordinal indicator (º), Right angle quotation mark (»), Fraction one quarter (1/4), Fraction one half (1/2), Fraction three quarters (3/4), and Inverted question mark (¿). Other characters include the Multiplication sign and the Division sign. In LATIN1, all GRAPHIC_IRV characters precede the non-GRAPHIC_IRV characters in the default collation, followed by the special symbols, followed by the accented capital letters, followed by the accented small letters. The Multiplication sign (×) is in the middle of the accented capital letters and the Division sign (÷) is in the middle of the accented small letters.
- LATIN1 is subject to the following conformance requirements, as specified in Clause 3, "Conformance", of ISO 8859-1:
 - A set of graphic characters is in conformance with ISO 8859-1 if it comprises all graphic characters specified therein to the exclusion of any other and if their coded representations are those specified by ISO 8859-1.
 - Equipment claimed to implement ISO 8859-1 shall implement all 191 characters.

4.2 Character strings

- ISO8BIT (or ASCII_FULL) specifies the name of a character repertoire that consists of all 255 characters, each consisting of exactly 8 bits, as specified in ISO 4873 and ISO 8859-1, including all control characters and all graphic characters except the character corresponding to the numeric value 0 (zero). The form-of-use is that corresponding to the coded representation of each character by a single 8-bit byte, with no designation escape sequences for other character sets. The default collating sequence is that corresponding to the bit combinations defined. The ISO8BIT character set is a superset of LATIN1 and, when restricted to the LATIN1 characters, produces the same collation and form-of-use.
- UTF16 and ISO10646 specify the name of a character repertoire that consists of every character represented by The Unicode Standard Version 2.0 and by ISO/IEC 10646 UTF-16, where each character is encoded using the UTF-16 encoding, occupying either 1 (one) or 2 octets.
- UTF8 specifies the name of a character repertoire that consists of every character represented by The Unicode Standard Version 2.0 and by ISO/IEC 10646 UTF-8, where each character is encoded using the UTF-8 encoding, occupying from 1 (one) through 6 octets.
- UCS2 specifies the name of a character repertoire that consists of every character represented by The Unicode Standard Version 2.0 and by ISO/IEC 10646 UCS2, where each character is encoded using the UCS2 encoding, in which each character occupies exactly 2 octets.
- SQL_TEXT specifies the name of a character repertoire that includes the <SQL language character>s and all other characters that are in character sets supported by the implementation. The SQL_TEXT character set is a superset of SQL_CHARACTER. The collation and form-of-use of SQL_TEXT is implementation-defined.
- SQL_IDENTIFIER specifies the name of a character repertoire that includes the <SQL language character>s and all other 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. The SQL_IDENTIFIER character set is a superset of SQL_CHARACTER but may be a subset of SQL_TEXT. The collation and form-of-use of SQL_IDENTIFIER is implementation-defined.
- The character sets SQL_CHARACTER, GRAPHIC_IRV (or ASCII_GRAPHIC), LATIN1, ISO8BIT (or ASCII_FULL), and UNICODE (or ISO10646) have both a “floor” and “ceiling” requirement to consist of exactly the characters specified. Any character data type associated with one of these character sets has an implied integrity constraint limiting a value of the data type to be a character string consisting only of characters from the specified character set. The SQL_TEXT and SQL_IDENTIFIER character sets have a similar “floor” requirement in that they must contain all characters that are in other character sets supported by the implementation (for SQL-data and for <identifier>s, respectively); however, SQL_TEXT and SQL_IDENTIFIER do not have a “ceiling” requirement.

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

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

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

4.3.1 Binary string comparison

All binary strings are mutually comparable. A binary string is identical to another binary string if and only if it is equal to that binary string in accordance with the comparison rules specified in Subclause 8.2, "<comparison predicate>".

4.3.2 Operations involving binary strings

4.3.2.1 Operators that operate on binary strings and return binary strings

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

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

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

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

4.3.2.2 Other operators involving binary strings

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

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

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

4.4 Bit strings

A bit string is a sequence of bits, each having the value of 0 (zero) or 1 (one). A bit string has a length, which is the number of bits in the string. The length is 0 (zero) or a positive integer.

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

- The name of the specific bit string data type (BIT or BIT VARYING).
- The length of the bit string data type (in bits).

4.4 Bit strings

4.4.1 Bit string comparison and assignment

All bit strings are mutually comparable. A bit string is identical to another bit string if and only if it is equal to that bit string in accordance with the comparison rules specified in Subclause 8.2, “<comparison predicate>”.

4.4.2 Operations involving bit strings

4.4.2.1 Operators that operate on bit strings and return bit strings

<bit concatenation> is an operator, ||, that returns the bit string made by concatenating the two bit string operands in the order given.

<bit substring function> is a triadic function identical in syntax and semantics to <character substring function> except that the first argument and the returned value are both bit strings.

4.4.2.2 Other operators involving bit strings

<length expression> returns the length (as an integer number of octets or bits according to the choice of function) of a given bit string.

<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.

4.5 Numbers

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

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

- The name of the specific numeric data type (NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, or DOUBLE PRECISION).
- The precision of the numeric data type.
- The scale of the numeric data type, if it is an exact numeric data type.
- An indication of whether the precision (and scale) are expressed in decimal or binary terms.

A value described by a numeric data type descriptor is always signed.

4.5.1 Characteristics of numbers

An exact numeric value has a precision and a scale. The precision is a positive integer that determines the number of significant digits in a particular radix (binary or decimal). The scale is a non-negative integer. A scale of 0 (zero) indicates that the number is an integer. For a scale of *S*, the exact numeric value is the integer value of the significant digits multiplied by 10^{-S} .

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 10^x , where x is the exponent.

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>".

4.5.2 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 4.2.2, "Operations involving character strings", and Subclause 4.4.2, "Operations involving bit strings") takes two strings as arguments and returns an integer.
- <length expression> (see Subclause 4.2.2, "Operations involving character strings", and Subclause 4.4.2, "Operations involving bit strings") operates on a string argument and returns an integer.
- <extract expression> (see Subclause 4.7.3, "Operations involving datetimes and intervals") operates on a datetime or interval argument and returns an integer.
- <cardinality expression> (see Subclause 4.11.3, "Operations involving collections") operates on a collection argument and returns an integer.

4.5 Numbers

- <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).

4.6 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 *unknown* truth value as the null value. This specification does not make a distinction between the null value of the boolean data type and the *unknown* truth value 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).

4.6.1 Comparison and assignment of booleans

All boolean data type values and SQL truth values are mutually comparable and assignable. 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.

4.6.2 Operations involving booleans

4.6.2.1 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 13, "Truth table for the AND boolean operator", and Table 14, "Truth table for the OR boolean operator").

4.6.2.2 Other operators involving booleans

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

4.7 Datetimes and intervals

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

A datetime data type descriptor contains:

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

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

An interval data type descriptor contains:

- The name of the interval data type (INTERVAL).
- An indication of whether the interval data type is a year-month interval or a day-time interval.
- The <interval qualifier> that describes the precision of the interval data type.

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

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

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

4.7.1 Datetimes

Table 4, “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 timezone field is in the subset, then both of them shall be included.

Table 4—Fields in datetime values

Keyword	Meaning
Primary datetime fields	
YEAR	Year
MONTH	Month within year
DAY	Day within month
HOURL	Hour within day
MINUTE	Minute within hour
SECOND	Second and possibly fraction of a second within minute
Timezone datetime fields	
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 mutually comparable only if they have the same <primary datetime field>s.

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

The surface of the earth is divided into zones, called time zones, in which every correct clock tells the same time, known as *local time*. Local time is equal to UTC (Coordinated Universal Time) plus the *time zone displacement*, which is an interval value that ranges between INTERVAL -'12:59' HOUR TO MINUTE and INTERVAL +'13:00' HOUR TO MINUTE. The time zone displacement is constant throughout a time zone, changing at the beginning and end of Daylight Time, where applicable.

A datetime value, of data type TIME WITHOUT TIME ZONE or TIMESTAMP WITHOUT TIME ZONE, may represent a local time, whereas a datetime value of data type TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE represents UTC. On occasion, UTC is adjusted by the omission of a second or the insertion of a "leap second" in order to maintain synchronization with sidereal time. This implies that sometimes, but very rarely, a particular minute will contain exactly 59, 61, or 62 seconds. Whether an SQL-implementation supports leap seconds, and the consequences of such support for date and interval arithmetic, is implementation-defined.

For the convenience of users, whenever a datetime value with time zone is to be implicitly derived from one without (for example, in a simple assignment operation), SQL assumes the value without time zone to be local, subtracts the default SQL-session time zone displacement 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 5, "Datetime data type conversions".

Table 5—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	<i>trivial</i>	<i>not sup- ported</i>	<i>not sup- ported</i>	Copy year, month, and day; set hour, minute, and second to 0 (zero)	$SV \Rightarrow TS_{w/oTZ}$ $\Rightarrow TS_{w/TZ}$
from TIME WITHOUT TIME ZONE	<i>not sup- ported</i>	<i>trivial</i>	$TV.UTC =$ $SV - STZD$ (modulo 24); $TV.TZ =$ $STZD$	Copy date fields from CURRENT_ DATE and time fields from TZ	$SV \Rightarrow TS_{w/oTZ}$ $\Rightarrow TS_{w/TZ}$
from TIME WITH TIME ZONE	<i>not sup- ported</i>	$SV.UTC +$ $SV.TZ$ (module 24)	<i>trivial</i>	$SV \Rightarrow TS_{w/oTZ}$ $\Rightarrow TS_{w/TZ}$	Copy date fields from CURRENT_ DATE and time and time zone fields from SV
from TIMESTAMP WITHOUT TIME ZONE	Copy date fields from SV	Copy time fields from TZ	$SV \Rightarrow$ $TS_{w/TZ}$ $\Rightarrow TS_{w/oTZ}$	<i>trivial</i>	$TV.UTC = SV$ $- STZD$; $TV.TZ = STZD$
from TIMESTAMP WITH TIME ZONE	$SV \Rightarrow$ $TS_{w/oTZ}$ $\Rightarrow DATE$	$SV \Rightarrow$ $TS_{w/oTZ}$ $\Rightarrow TIME_{w/oTZ}$	Copy time and time zone fields from SV	$SV.UTC +$ $SV.TZ$	<i>trivial</i>

4.7.2 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 6, “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 6—Fields in year-month INTERVAL values

Keyword	Meaning
YEAR	Years
MONTH	Months

Table 7, “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 7—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 8, “Valid values for fields in INTERVAL values”, specifies the constraints on subsequent field values.

Table 8—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 mutually 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 mutually 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.

4.7.3 Operations involving datetimes and intervals

Table 9, "Valid operators involving datetimes and intervals", specifies the declared types of arithmetic expressions involving datetime and interval operands.

Table 9—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 mutually comparable. Operations involving values of type interval require that the interval values be mutually 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 part, then the local time zone is effectively used.

<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.

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

4.8 User-defined types

4.8 User-defined types

A user-defined type is a schema object, identified by a <user-defined type name>. The definition of a user-defined type specifies a number of components, including in particular 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 representation of a user-defined type is expressed either as a single data type (some predefined data type, called the *source type*), in which case the user-defined type is said to be a *distinct type*, or as a list of attribute definitions, in which case it is said to be a *structured type*.

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>. Each <original method specification> specifies the <method name>, the <specific name>, the <SQL parameter declaration list>, the <returns data type>, the <result cast from type> (if any), the <transform group specification> (if any), whether the method is type-preserving, the <language clause>, the <parameter style> if the language is not SQL, whether STATIC is specified, whether the method is deterministic, to what extent the method accesses SQL (possibly writes SQL data, possibly reads SQL data, possibly contains SQL, or does not possibly contain SQL), and whether the method should be evaluated as NULL whenever any argument is NULL, without actually invoking the method.

Each <overriding method specification> specifies the <method name>, the <specific name>, the <SQL parameter declaration list> and the <returns data type>. For each <overriding method specification>, there must 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 must 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*.

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. 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.
- An indication of whether the user-defined type is ordered.
- 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 reference type for which the structured type is the referenced type has a user-defined representation, a derived representation, or a system-defined representation, and the list of attributes of the derived representation.
NOTE 5 – “user-defined representation”, “derived representation”, and “system-defined representation” of a reference type are defined in Subclause 4.10, “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 <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 is specified.
 - An indication whether the method is deterministic.
 - An indication whether the method possibly writes SQL data, possibly reads SQL data, possibly contains SQL, or does not possibly contain SQL.

4.8 User-defined types

- 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 6 – 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>.

4.8.1 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 $\text{FUNCTION } A(T)$ and its result data type is AT . The signature of the mutator function for this attribute is $\text{FUNCTION } A(T \text{ RESULT}, AT)$ 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 $A(MV) = AV$ and for every attribute A' ($A' \neq A$) of T , $A'(MV) = A'(V)$. The most specific type of MV is the most specific type of V .

4.8.2 Constructors

Associated with every structured type ST is at least one *constructor function*, implicitly defined when ST is defined. The constructor function is defined if and only if ST is instantiable.

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

4.8.3 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 11.40, "<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 7 – Because a subtype family has exactly one maximal supertype, if two types have a common subtype, they must 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 must 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. However, a type can have at most 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.

4.8.4 User-defined type comparison and assignment

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

- The type designator of T_b is in the type precedence list of T_a .
- The user-defined type descriptor of T_b includes an ordering form that is EQUALS or FULL.
- 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 .

4.8 User-defined types

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 $V2$ of whose declared 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 — they must be the same throughout a type family. If the comparison category is either STATE or RELATIVE, then the comparison functions of $T1$ and $T2$ are constrained to be equivalent; if the comparison category is MAP, they are not constrained to be equivalent.

NOTE 8 — 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 9 — “Subtype” and “subtype family” are defined in Subclause 4.8.3, “Subtypes and supertypes”.

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 $UDT2$.

4.8.5 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.

4.9 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 $FD1_n$ is compatible with the <data type> of $FD2_n$.

4.10 Reference types

A *REF value* is a value that references a row in a *referenceable table* (see Subclause 4.16.2, "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*. T is the *referenced type* of RT .

A REF value is represented as an octet sequence with an implementation-defined length.

Values of two reference types are mutually 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 <dereference operator> on the value at RS . A scope is specified as table name and consists at any time of every site that is occupied by a row in that table.

A reference type is described by a reference type descriptor. A reference type descriptor includes:

- The name of the referenceable table, if any, that is the scope of the reference type.
- The name of the referenced type.

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 $RT1$.

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.

4.10 Reference types

4.10.1 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 6.14, “<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 6.15, “<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 6.10, “<method reference>”).

4.11 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 types (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 one collection type, arrays.

A specific <collection type> *CT* is a <data type> specified by pairing a specific <collection type constructor> *CC* with a specific data type *EDT*. 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*. *CC* specifies the type of collection, such as ARRAY, that every value of *CT* is, and thus determines the operators that are available for operating on or returning values of *CT*.

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

- The descriptor of the element type of *CT*.
- An indication of the type of the collection *CT*: ARRAY.

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

4.11.1 Arrays

An *array* is a collection *A* in which each element is associated with exactly one ordinal position in *A*. If *n* is the cardinality of *A*, then the ordinal position *p* of an element is an integer in the range 1 (one) $\leq p \leq n$. If *EDT* is the element type of *A*, then *A* can thus be considered as a function of the integers in the range 1 (one) to *n* onto *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 the same array if and only if $A1$ and $A2$ have the same cardinality n and if, for all i in the range $1 \text{ (one)} \leq i \leq n$, the element at ordinal position i in $A1$ is the same as 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 $0 \text{ (zero)} \leq j < n2$ such that, for every i in the range $1 \text{ (one)} \leq i \leq n1$, the element at ordinal position i in $A1$ is the same as the element at ordinal position $i+j$ in $A2$.

4.11.2 Collection comparison

Two collections are comparable if and only if they are of the same collection type and their element types are comparable.

Each collection type has a defined *element order*. Comparisons may be defined in terms of the element order of the collections. The element order defines the pairs of corresponding elements from the collections being compared. The element order of an array is implicitly defined by the ordinal position of its elements.

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

4.11.3 Operations involving collections

4.11.3.1 Operators that operate on array values and return array elements

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

4.11.3.2 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.

4.12 Type conversions and mixing of data types

Values of the data types NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, and DOUBLE PRECISION are numbers and are all mutually comparable and mutually assignable. If an assignment would result in a loss of the most significant digits, 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 generally governed by Subclause 6.26, "<numeric value expression>".

Values corresponding to the data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are mutually assignable if and only if they are taken from the same character repertoire. If they are from different character repertoires, then the value of the source of the assignment must be translated to the character repertoire of the target before an assignment is possible. Such translation may be implementation-defined and implicitly performed, in which case the two character data types are also mutually assignable. 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 would result in the loss of characters due to truncation, then a warning

4.12 Type conversions and mixing of data types

condition is raised. The values are mutually comparable only if they are mutually assignable and can be coerced to have the same collation. The comparison of two character strings depends on the collating sequence used for the comparison (see Table 3, "Collating sequence usage for comparisons"). When values of unequal length are compared, if the collating sequence for the comparison has the NO PAD characteristic and the shorter value is equal to a prefix of the longer value, then the shorter value is considered less than the longer value. If the collating sequence 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.

Values corresponding to the binary data type are mutually assignable. 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. When binary string values are compared, they must have exactly the same length (in octets) to be considered equal. Binary string values can only be compared for equality.

Values corresponding to the data types BIT and BIT VARYING are always mutually comparable and are mutually assignable. If a store assignment would result in the loss of bits due to truncation, then an exception condition is raised. If a store assignment to a fixed-length bit string would result in the addition of bits, then an exception condition is raised. If a retrieval assignment would result in the loss of bits due to truncation, then a warning condition is raised. When values of unequal length are compared, if the shorter is a prefix of the longer, then the shorter is less than the longer; otherwise, the longer is effectively truncated to the length of the shorter for the purposes of comparison. When values of equal length are compared, then a bit-by-bit comparison is made. A 0-bit is less than a 1-bit.

Values corresponding to the data type boolean are always mutually comparable and are mutually assignable.

Values of type datetime are mutually assignable 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).

Values of type interval are mutually assignable only if the source and target of the assignment are both year-month intervals or if they are both day-time intervals.

Values corresponding to user-defined types are discussed in Subclause 4.8.4, "User-defined type comparison and assignment".

Values corresponding to reference types are discussed in Subclause 4.10, "Reference types".

Values corresponding to the collection types are discussed in Subclause 4.11, "Collection types".

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.

Values corresponding to row types are mutually assignable if and only if both have the same degree and every field in one row type is mutually assignable to the field in the same ordinal position of the other row type. Values corresponding to row types are mutually comparable if and only if both have the same degree and every field in one row type is mutually comparable to the field in the same ordinal position of the other row type.

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

4.12 Type conversions and mixing of data types

NOTE 10 – 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.

4.13 Data conversions

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 6.22, “<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 one or more user-defined types 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 a 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*.

A 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>, or <update statement: searched>.
- *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 a SQL-invoked function that is invoked during the execution of *S*.

4.14 Domains**4.14 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 <collation name> from the <collate clause>, if any, 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.

4.15 Columns, fields, and attributes

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

A table (see Subclause 4.16, “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 11 – 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. The possible values of nullability characteristic are *known not nullable* and *possibly nullable*.

A column *C* of a base table *T* has a nullability characteristic that 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 value of *C* can never be null through some additional implementation-defined rule or rules.

A column *C* of a derived table is *known not nullable*, according to the Syntax Rules of Subclause 7.7, "<joined table>", Subclause 7.11, "<query specification>", and Subclause 7.12, "<query expression>".

Otherwise, a column *C* is *possibly nullable*.

NOTE 12 – Whether a column of a virtual table is possibly nullable or known not nullable is specified in the Syntax Rules of various Subclauses, including Subclause 7.7, "<joined table>", Subclause 7.11, "<query specification>", and Subclause 7.12, "<query expression>".

A column, field, or attribute may be defined with a data type that is a reference type. For such a column, field, or attribute, if the reference type specifies a <scope clause>, then the user may specify whether reference values must be checked. If the user has specified that reference values must be checked, then:

- Whenever the value of the column, field, or attribute is changed, the new value of the column, field, or attribute must reference a row of one of the tables in the <scope clause> of the reference type.
- The column, field, or attribute definition may specify that, whenever a row is deleted from one of the tables in the <scope clause>, either the value of the (referencing) column, field, or attribute is set to the null value, or that such deletions are prohibited; in the latter case, attempts to perform such deletions result in the raising of an exception condition: *constraint violation*.

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 <collation name> from the <collate clause>, if any, 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 the column is a self-referencing column of a base table or not.

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 <collation name> from the <collate clause>, if any, 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*.
- If the data type of the attribute is a reference type, then whether reference values must be checked.

4.15 Columns, fields, and attributes

— 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 <collation name> from the <collate clause>, if any, of *F*.
- The ordinal position of *F* within the row type that simply contains it.
- If the data type of the field is a reference type, then whether reference values must be checked.

4.16 Tables

A table is a collection of rows having one or more columns. A row is an instance 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.

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*.

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

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

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

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

All base tables are *updatable*. Derived tables are either updatable or not updatable. The operations of update and delete are permitted for updatable tables, subject to constraining Access Rules. Some updatable tables, including all base tables, are also *insertable-into*, in which case the operation of insert is also permitted, again subject to Access Rules.

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.

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 base 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 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 DELETE ROWS was specified or ON COMMIT PRESERVE 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 insertable-into 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.

4.16 Tables

4.16.1 Types of tables

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

The *simply underlying tables* of a derived table or cursor are defined in Subclause 7.11, "<query specification>", Subclause 7.12, "<query expression>", and Subclause 14.1, "<declare cursor>". A <table or query name> has no simply underlying tables.

The *underlying tables* 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 tables* 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 tables* 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*, defined as follows:

- If *TORQN* identifies a base table, 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 tables* 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.

A *grouped table* is a set of groups derived during the evaluation of a <group by clause>. A group *G* is a multiset of rows in which, for every grouping column *GC*, if the value of *GC* in some row is *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 equal to the value of *GC* in *R2* (or both are the null value), 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 declared local temporary table is a module local temporary table. 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 that is declared in an SQL-client module is a named table defined by a <temporary table declaration> that is effectively materialized the first time any <externally-invoked procedure> in the <SQL-client module definition> that contains the <temporary table declaration> is executed. A declared local temporary table is accessible only by <externally-invoked procedure>s in the <SQL-client module definition> that contains the <temporary table declaration>. 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.

4.16.2 Referenceable tables, subtables, and supertables

A table BT 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 BT ; 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 BT is a base table, then the table constraint "UNIQUE($REFCN$)" is implicit in the definition of BT . 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 base 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 must be created on a structured type and the structured type of T_a must 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 .

4.16 Tables

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 must 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.

4.16.3 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>.

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

A <table reference> that satisfies certain properties specified in this international standard can be used to designate an *updatable table*. Certain table updating operations, specified by SQL-data change statements, are available in connection with updatable tables. The value of an updatable table T is determined by the result of the mostly recently executed SQL-data change statement (see Subclause 4.30.2, “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 effect of deleting a row R from T is to replace the value TV of T by the result of the <query expression>:

```
SELECT * FROM T EXCEPT ALL VALUES RR
```

where RR is some <row value constructor> whose value is R . The primary effect of a <delete statement: positioned> on T is to delete exactly one specified row from T . The primary effect of a <delete statement: searched> on T is to delete zero or more rows from T .

The effect of replacing a row R in T is to replace the value TV of T by the result of the <query expression>:

```
SELECT *
FROM T
EXCEPT ALL
VALUES RR
UNION ALL
VALUES RR1
```

where RR is some <row value constructor> whose value is R and $RR1$ is some <row value constructor> whose value is the row to replace R in T . The primary effect of an <update statement: positioned> on T is to replace exactly one specified row in T with some specified row. The primary effect of an <update statement: searched> on T is to replace zero or more rows in T .

If T , as well as being updatable, is insertable-into, then rows can be inserted into it. The effect of inserting a row R into T is to replace the value TV of T by the result of the <query expression>:

```
SELECT * FROM T UNION ALL VALUES RR
```

where RR is some <row value constructor> whose value is R . 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.

Each of the table updating operations, when applied to T , can have various secondary effects, as specified in ISO/IEC 9075. 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 GLOBAL CHECK OPTION, which might result in alteration or reversal of primary effects.
- 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 .

The secondary effects of table updating operations on T on proper supertables and subtables of T are as follows:

- When row R is deleted from T , for every table ST that is a proper supertable or proper subtable of T , the corresponding superrow or subrow SR of R in ST is deleted from ST .
- When row R is replaced in T , for every table ST that is a proper supertable or a proper subtable of T the corresponding superrow or subrow SR of R in ST is replaced in ST .
- When row R is inserted into T , for every proper supertable ST of T the corresponding superrow SR of R is inserted into ST .

4.17 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 either a table constraint, a domain constraint or an assertion. A constraint is described by a constraint descriptor. A constraint descriptor is either a table constraint descriptor, a domain constraint descriptor or an assertion descriptor. Every constraint descriptor includes:

- The name of the constraint.
- An indication of whether or not the constraint is deferrable.
- An indication of whether the initial constraint mode is *deferred* or *immediate*.

A <query expression> or <query specification> is *possibly non-deterministic* if an SQL-implementation might, at two different times where the state of the SQL-data is the same, produce results that differ by more than the order of the rows due to General Rules that specify implementation-dependent behavior.

No integrity constraint shall be defined using a <query specification> or a <query expression> that is possibly non-deterministic.

4.17.1 Checking of constraints

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

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

NOTE 13 – This includes SQL-statements that are executed as a direct result or an indirect result of executing a different SQL-statement.

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

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

4.17.2 Table constraints

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

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

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

- An indication of whether it was defined with PRIMARY KEY or UNIQUE.
- The names and positions of the *unique columns* specified in the <unique column list>.

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

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

- The names of the *referencing columns* specified in the <referencing columns>.
- The names of the *referenced columns* and *referenced table* specified in the <referenced table and columns>.
- The value of the <match type>, if specified, and the <referential triggered actions>, if specified.

NOTE 14 – 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 option> were specified.

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

- The <search condition>.

A 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 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 option> 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*.

4.17 Integrity constraints

- 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> is not false for any row of a table.

4.17.3 Domain constraints

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

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

- The <search condition>.

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

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

4.17.4 Assertions

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

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

- The <search condition>.

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

4.18 Functional dependencies

This Subclause defines *functional dependency* and specifies a minimal set of rules that a conforming implementation must 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.

4.18.1 General rules and definitions

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

Let " $T: A \mapsto B$ " (read "in T , A determines B " or " B is functionally dependent on A in T ") denote the functional dependency of B on A in T , which is true if, for any possible value of T , any two rows that agree in value for every column in A also agree in value for every column in B . Two rows agree in value for a column if the two values either are both null or compare as equal under the General Rules of Subclause 8.2, "<comparison predicate>". When the table T is understood from context, the abbreviation " $A \mapsto B$ " may also be used.

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

Let $A \mapsto B$ and $C \mapsto D$ be any two functional dependencies in T . The following are also functional dependencies in T :

— $A \text{ UNION } (C \text{ DIFFERENCE } B) \mapsto B \text{ UNION } D$

— $C \text{ UNION } (A \text{ DIFFERENCE } D) \mapsto B \text{ UNION } D$

NOTE 15 – 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 $X \mapsto X$ for any X . A functional dependency of this form is an axiomatic functional dependency, as is $X \mapsto Y$ where Y is a subset of X . $X \mapsto Y$ is a non-axiomatic functional dependency if Y is not a subset of X .

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 16 – "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 $S2$, $S1 \mapsto S2$, and $S2$ is a BUC-set, then $S1$ is also a BUC-set.

NOTE 17 – A BUC-set may be empty, in which case there is at most one row in the table. This case must 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 18 – “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 $S2$, $S1 \mapsto S2$, and $S2$ is a BPK-set, then $S1$ is also a BPK-set.

NOTE 19 – 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 expression>s that are the operands of AC . Suppose that both $RVE1$ and $RVE2$ are <row value constructor>s. Let n be the number of <row value constructor element>s in $RVE1$. Let $RVEC1_i$ and $RVEC2_i$, $1 \text{ (one)} \leq i \leq n$, be the i -th <row value constructor element> of $RVE1$ and $RVE2$, respectively. The <comparison predicate> “ $RVEC1_i = RVEC2_i$ ” is called an *equality AND-component* of SC .

4.18.2 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 $UCL \mapsto CT$ is a *known functional dependency* in T .

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

4.18.3 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>).

4.18.4 Known functional dependencies in a <joined table>

Let $T1$ and $T2$ denote the tables identified by the first and second <table reference>s of some <joined table> JT . Let R denote the table that is the result of JT . Let CT be the set of columns of the result of JT .

Every column of R has some counterpart in either $T1$ or $T2$. If NATURAL is specified or the <join specification> is a <named columns join>, then some columns of R may have counterparts in both $T1$ and $T2$.

A set of columns S of R is a *BPK-set* if S has some counterpart in $T1$ or $T2$ that is a BPK-set, every member of S is known not null, and $S \mapsto CT$ is a *known functional dependency* of R .

A set of columns S of R is a *BUC-set* if S has some counterpart in $T1$ or $T2$ that is a BUC-set, every member of S is known not null, and $S \mapsto CT$ is a *known functional dependency* of R .

NOTE 20 – The following rules for known functional dependencies in a <joined table> are not mutually exclusive. The set of known functional dependencies is the union of those dependencies generated by all applicable rules, including the rules of deduction presented earlier.

If <join condition> is specified, AP is an equality AND-component of the <search condition>, one comparand of AP is a column reference CR , and the other comparand of AP is a <literal>, then let CRC be the counterparts of CR in R . Let $\{\}$ denote the empty set. $\{\} \mapsto \{CRC\}$ is a *known functional dependency* in R if any of the following conditions is true:

- INNER is specified.
- If LEFT is specified and CR is a column reference to a column in $T1$.
- If RIGHT is specified and CR is a column reference to a column in $T2$.

NOTE 21 – An SQL-implementation may also choose to recognize $\{\} \rightarrow \{CRC\}$ 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 . $\{CRAC\} \mapsto \{CRBC\}$ 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 22 — An SQL-implementation may also choose to recognize the following as known functional dependencies: $\{CRA\} \mapsto \{CRBC\}$ 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 23 — An SQL-implementation may also choose to recognize similar known functional dependencies of the form $\{CRA_1, \dots, CRA_N\} \mapsto \{CRCB\}$ in case one comparand is a deterministic expression of column references $CRA_1, \dots, CRA_N$ under similar conditions.

If NATURAL is specified, or if a <join specification> immediately containing a <named columns join> is specified, then let $C_1, \dots, C_N$ 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 $\{ \text{COALESCE} (TN1.C_i, TN2.C_i) \} \mapsto \{ TN1.C_i \}$, for all i between 1 (one) and N .
- If INNER or RIGHT is specified, then $\{ \text{COALESCE} (TN1.C_i, TN2.C_i) \} \mapsto \{ TN2.C_i \}$, for all i between 1 (one) and N .

The *known functional dependencies* of R are the known functional dependencies of:

```
SELECT SL FROM IR
```

4.18.5 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 or query name> TQN (with or without ONLY), then the counterparts of the BPK-sets and BUC-sets of TQN are the BPK-sets and BUC-sets, respectively, of R . If $A \mapsto B$ is a functional dependency in the result of TQN , and AC and BC are the counterparts of A and B , respectively, then $AC \mapsto BC$ is a *known functional dependency* in R .

- If TR immediately contains a <derived table> DT , then the counterparts of the BPK-sets and BUC-sets of DT are the BPK-sets and BUC-sets, respectively, of R . If $A \mapsto B$ is a functional dependency in the result of DT , and AC and BC are the counterparts of A and B , respectively, then $AC \mapsto BC$ is a *known functional dependency* in R .
- If TR immediately contains a <joined table> JT , then the counterparts of the BPK-sets and BUC-sets of JT are the BPK-sets and BUC-sets, respectively, of R . If $A \mapsto B$ is a functional dependency in the result of JT , and AC and BC are the counterparts of A and B , respectively, then $AC \mapsto BC$ is a *known functional dependency* in R .
- If TR immediately contains a <lateral derived table> LDT , then the counterparts of the BPK-sets and BUC-sets of LDT are the BPK-sets and BUC-sets, respectively, of R . If $A \mapsto B$ is a functional dependency in the result of LDT , and AC and BC are the counterparts of A and B , respectively, then $AC \mapsto BC$ is a *known functional dependency* in R .
- If TR immediately contains a <collection derived table> CDT , and WITH ORDINALITY is specified, then let $C1$ and $C2$ be the two columns names of CDT . $\{C2\}$ is a BPK-set and a BUC-set, and $\{C2\} \mapsto \{C2, C1\}$ is a *known functional dependency*. If WITH ORDINALITY is not specified, then these rules do not identify any BPK-set, BUC-set, or non-axiomatic known functional dependency.

4.18.6 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 .

4.18.7 Known functional dependencies in the result of a <where clause>

Let T be the table that is the operand of the <where clause>. Let R be the result of the <where clause>. A set of columns S in R is a *BUC-set* if there is a <table reference> TR such that every member of S has a counterpart in TR , the counterpart of S in TR is a BUC-set, and $S \mapsto CR$, where CR is the set of all columns of R . If, in addition, the counterpart of S is a BPK-set, then S is a *BPK-set*.

If $A \mapsto B$ is a known functional dependency in T , then let AC be the set of columns of R whose counterparts are in A , and let BC be the set of columns of R whose counterparts are in B . $AC \mapsto BC$ is a *known functional dependency* in R .

If AP is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of AP is a column reference CR , and the other comparand of AP is a <literal>, then let CRC be the counterpart of CR in R . $\{\} \mapsto \{CRC\}$ is a *known functional dependency* in R , where $\{\}$ denotes the empty set.

NOTE 24 – An SQL-implementation may also choose to recognize $\{\} \mapsto \{CRC\}$ as a known functional dependency if the other comparand is a deterministic expression containing no column references.

If AP is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of AP is a column reference CRA , and the other comparand of AP is a column reference CRB , then let $CRAC$ and $CRBC$ be the counterparts of CRA and CRB in R . $\{CRBC\} \mapsto \{CRAC\}$ and $\{CRAC\} \mapsto \{CRBC\}$ are *known functional dependencies* in R .

NOTE 25 – An SQL-implementation may also choose to recognize known functional dependencies of the form $\{CRAC_1, \dots, CRAC_N\} \mapsto \{CRBC\}$ if one comparand is a deterministic expressions that contains column references $CRA_1, \dots, CRA_N$ and the other comparand is a column reference CRB .

4.18.8 Known functional dependencies in the result of a <group by clause>

Let $T1$ be the table that is the operand of the <group by clause>, and let R be the result of the <group by clause>.

Let G be the set of columns specified by the <grouping column reference list> of the <group by clause>, after applying all syntactic transformations to eliminate ROLLUP, CUBE, and GROUPING SETS.

The columns of R are the columns of G , with an additional column CI , whose value in any particular row of R somehow denotes the subset of rows of $T1$ that is associated with the combined value of the columns of G in that row.

If every element of G is a column reference to a known not null column, then G is a *BUC-set* of R . If G is a subset of a *BPK-set* of columns of $T1$, then G is a *BPK-set* of R .

$G \mapsto CI$ is a *known functional dependency* in R .

NOTE 26 – Any <set function specification> that is specified in conjunction with R is necessarily a function of CI . If $SFVC$ denotes the column containing the results of such a <set function specification>, then $CI \mapsto SFVC$ holds true, and it follows that $G \mapsto SFVC$ is a *known functional dependency* in the table containing $SFVC$.

4.18.9 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> directly 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 .

4.18.10 Known functional dependencies in a <query specification>

Let T be the <table expression> simply contained in the <query specification> and let R be the result of the <query specification>.

Let SL be the <select list> of the <query specification>.

Let $T1$ be T extended to the right with columns arising from <value expression>s contained in the <select list>, as follows: A <value expression> VE that is not a column reference specifies a computed column CC in $T1$. For every row in $T1$, the value in CC is the result of VE .

Let S be a set of columns of R such that every element of S arises from the use of <asterisk> in SL or by the specification of a column reference as a <value expression> simply contained in SL . S has counterparts in T and $T1$. If the counterpart of S in T is a BPK-set, then S is a *BPK-set*. If the counterpart of S in T is a BUC-set or a BPK-set, then S is a *BUC-set*.

If $A \mapsto B$ is some known functional dependency in T , then $A \mapsto B$ is a *known functional dependency* in $T1$.

Let CC be the column specified by some <value expression> VE in the <select list>.

If $OP1, OP2, \dots$ are the operands of VE that are column references, then $\{OP1, OP2, \dots\} \mapsto CC$ is a *known functional dependency* in $T1$.

Let $C \mapsto D$ be some known functional dependency in $T1$. If all the columns of C have counterparts in R , then let DR be the set comprising those columns of D that have counterparts in R . $C \mapsto DR$ is a *known functional dependency* in R .

4.18.11 Known functional dependencies in a <query expression>

If a <with clause> is specified, and RECURSIVE is not specified, then the *BPK-sets*, *BUC-sets*, and *known functional dependencies* of the table identified by a <query name> in the <with list> are the same as the *BPK-sets*, *BUC-sets*, and *known functional dependencies* of the corresponding <query expression>, respectively. If RECURSIVE is specified, then the *BPK-sets*, *BUC-sets*, and non-axiomatic *known functional dependencies* are implementation-defined.

A <query expression> that is a <query term> that is a <query primary> that is a <simple table> or a <joined table> is covered by previous Subclauses of this Clause.

If the <query expression> specifies UNION, EXCEPT or INTERSECT, then let $T1$ and $T2$ be the left and right operand tables and let R be the result. Let CR be the set comprising all the columns of R .

Each column of R has a counterpart in $T1$ and a counterpart in $T2$.

Case:

- If EXCEPT is specified, then a set S of columns of R is a *BPK-set* if its counterpart in $T1$ is a *BPK-set*. S is a *BUC-set* if its counterpart in $T1$ is a *BUC-set*.
- If UNION is specified, then there are no *BPK-sets* and no *BUC-sets*.
- If INTERSECT is specified, then a set S of columns of R is a *BPK-set* if either of its counterparts in $T1$ and $T2$ is a *BPK-set*. S is a *BUC-set* if either of its counterparts in $T1$ and $T2$ is a *BUC-set*.

Case:

- If UNION is specified, then no non-axiomatic functional dependency in $T1$ or $T2$ is a *known functional dependency* in R , apart from any functional dependencies determined by implementation-defined rules.
- If EXCEPT is specified, then all *known functional dependencies* in $T1$ are *known functional dependencies* in R .

- If INTERSECT is specified, then all known functional dependencies in $T1$ and all known functional dependencies in $T2$ are *known functional dependencies* in R .

NOTE 27 – Other known functional dependencies may be determined according to implementation-defined rules.

4.19 Candidate keys

If the functional dependency $CK \mapsto CT$ holds true in some table T , where CT consists of all columns of T , and there is no proper subset $CK1$ of CK such that $CK1 \mapsto CT$ holds true in T , then CK is a *candidate key* of T . The set of candidate keys SCK is nonempty because, if no proper subset of CT is a candidate key, then CT is a candidate key.

NOTE 28 – Because a candidate key is a set (of columns), SCK is therefore a set of sets (of columns).

A candidate key CK is a *strong candidate key* if CK is a BUC-set, or if T is a grouped table and CK is a subset of the set of grouping columns of T . Let $SSCK$ be the set of strong candidate keys.

Let PCK be the set of P such that P is a member of SCK and P is a *BPK-set*.

Case:

- If PCK is nonempty, then the *primary key* is chosen from PCK as follows: If PCK has exactly one element, then that element is the primary key; otherwise, the left-most element of PCK is chosen according to the “left-most rule” below. The primary key is also the *preferred candidate key*.
- Otherwise, there is no primary key and the *preferred candidate key* is chosen as follows:

Case:

- If $SSCK$ has exactly one element, then it is the preferred candidate key; otherwise, if $SSCK$ has more than one element, then the left-most element of $SSCK$ is chosen, according to the “left-most” rule below.
- Otherwise, if SCK has exactly one element, then it is the preferred candidate key; otherwise, the left-most element of SCK is chosen, according to the “left-most” rule below.

- The “left-most” rule:

- This rule uses the ordering of the columns of a table, as specified elsewhere in this part of ISO/IEC 9075.

To determine the left-most of two sets of columns of T , first list each set in the order of the column-numbers of its members, extending the shorter list with zeros to the length of the longer list. Then, starting at the left of each ordered list, step forward until a pair of unequal column numbers, one from the same position in each list, is found. The list containing the number that is the smaller member of this pair identifies the left-most of the two sets of columns of T .

To determine the left-most of more than two sets of columns of T , take the left-most of any two sets, then pair that with one of the remaining sets and take the left-most, and so on until there are no remaining sets.

4.20 SQL-schemas

An SQL-schema is a persistent descriptor that includes:

- The <schema name> of the SQL-schema.
- The <authorization identifier> of the owner of the SQL-schema.
- The <character set 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. Each component descriptor is either a domain descriptor, a base table descriptor, a view descriptor, a constraint descriptor, a privilege descriptor, a character set descriptor, a collation descriptor, a translation descriptor, a user-defined type descriptor, or a routine descriptor. 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.

A schema is created initially using a <schema definition> and may be subsequently modified incrementally over time by the execution of <SQL schema statement>s. <schema name>s are unique within a catalog.

A <schema name> is explicitly or implicitly qualified by a <catalog name> that identifies a catalog.

Base tables and views are identified by <table name>s. A <table name> consists of a <schema name> and an <identifier>. For a persistent table, the <schema name> identifies the schema in which the base table or view identified by the <table name> was defined. Base tables and views defined in different schemas can have <identifier>s that are equal according to the General Rules of Subclause 8.2, “<comparison predicate>”.

If a reference to a <table name> does not explicitly contain a <schema name>, then a specific <schema name> is implied. The particular <schema name> associated with such a <table name> depends on the context in which the <table name> appears and is governed by the rules for <schema qualified name>.

If a reference to an SQL-invoked routine that is contained in a <routine invocation> does not explicitly contain a <schema name>, then the SQL-invoked routine is selected from the SQL-path of the schema.

The *containing schema* of an <SQL schema statement> is defined as the schema identified by the <schema name> implicitly or explicitly contained in the name of the object that is created or manipulated by that SQL-statement.

4.21 SQL-client modules

An *SQL-client module* is an object specified in the module language. SQL-client modules are created and destroyed by implementation-defined mechanisms (which can include the granting and revoking of module privileges). SQL-client modules exist in the SQL-environment containing an SQL-client. The <externally-invoked procedure>s of an SQL-client module are invoked by host language programs. The <language clause> of an SQL-client module specifies a host programming

language. The format of an SQL-client module is specified by <SQL-client module definition> (see Subclause 13.1, "<SQL-client module definition>").

An SQL-client module consists of:

- An <SQL-client module name>.
- A <language clause>.
- A <module authorization clause> with either or both of a <module authorization identifier> and a <schema name>.
- An optional SQL-path 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 <user-defined type>s that are contained in the SQL-client module.
- An optional <module character set specification> that identifies the character set used to express the SQL-client module.

NOTE 29 – 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.
- Zero or more <temporary table declaration>s.
- Zero or more cursors specified by <declare cursor>s.
- One or more <externally-invoked procedure>s.

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.

4.22 Externally-invoked procedures

Externally-invoked routines are always procedures ("externally-invoked procedure") that are invoked by "call" statements in compilation units of the specified standard programming language. Externally-invoked procedures are always SQL routines. Externally-invoked procedures are specified with a <language clause> that specifies any language other than LANGUAGE SQL.

4.23 SQL-invoked routines

An *SQL-invoked routine* is an SQL-invoked procedure or an SQL-invoked function. An SQL-invoked routine comprises at least a <schema qualified routine name>, a sequence of <SQL parameter declaration>s, and a <routine body>.

An SQL-invoked routine is an element of an SQL-schema and is called a *schema-level routine*.

An SQL-invoked routine *SR* is said to be *dependent* on a user-defined type *UDT* if and only if *SR* is created during the execution of the <user-defined type definition> that created *UDT*. An SQL-invoked routine that is dependent on a user-defined type may not be destroyed by a <drop routine statement>. It is destroyed implicitly by a <drop data type statement>.

A <predicate> *P* is said to be dependent on an SQL-invoked routine *SR* if and only if *SR* is the ordering function included in the user-defined descriptor of a user-defined type *UDT* and one of the following conditions is true:

- *P* is a <comparison predicate> that immediately contains a <row value expression> whose declared type is some user-defined type *T1* whose comparison type is *UDT*.
- *P* is a <quantified comparison predicate> that immediately contains a <row value expression> that has some field whose declared type is some user-defined type *T1* whose comparison type is *UDT*.
- *P* is a <unique predicate> that immediately contains a <table subquery> that has a column whose declared type is some user-defined type *T1* whose comparison type is *UDT*.
- *P* is a <match predicate> that immediately contains a <row value expression> that has some field whose declared type is some user-defined type *T1* whose comparison type is *UDT*.
- *P* is a <comparison predicate> with some corresponding value whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UDT*.
- *P* is a <quantified comparison predicate> that immediately contains a <row value expression> that has some field whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UDT*.
- *P* is a <unique predicate> that immediately contains a <table subquery> that has a column whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UDT*.
- *P* is a <match predicate> that immediately contains a <row value expression> that has some field whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UDT*.

NOTE 30 – “Comparison type” is defined in Subclause 4.8.4, “User-defined type comparison and assignment”.

A <set function specification> *SFS* is said to be *dependent* on an SQL-invoked routine *SR* if and only if all the following are true:

- *SR* is the ordering function included in the user-defined descriptor of a user-defined type *UDT*.
- *SFS* is a <general set function> whose <set function type> *SFS* is MAX or MIN or *SFS* is a <general set function> whose <set qualifier> is DISTINCT.

— The declared type of the <value expression> of *SFS* is *UDT*.

A <group by clause> *GBC* is said to be *dependent* on an SQL-invoked routine *SR* if and only if all the following are true:

- *SR* is the ordering function included in the user-defined descriptor of a user-defined type *UDT*.
- The declared type of a grouping column of *GBC* is *UDT*.

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 11.49, "<SQL-invoked routine>").

An *SQL-invoked function* is an SQL-invoked routine whose invocation returns a value. Every parameter of an SQL-invoked function is an input 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 11.49, "<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 result data type of a type-preserving function is some subtype of the data type of its result SQL parameter.

An *SQL-invoked method* is an SQL-invoked function that is specified by <method specification designator> (see Subclause 11.49, "<SQL-invoked routine>"). There are two kinds of SQL-invoked methods: *instance SQL-invoked methods* and *static SQL-invoked methods*. All SQL-invoked methods are associated with a structured type, also known as the *type of the method*. The <routine 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. An instance SQL-invoked method satisfies 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.

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 11.49, "<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".

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 schema statement>*, *<SQL connection statement>*, or *<SQL transaction statement>*.

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.

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 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 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 <trigger 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 same return value for a given list of SQL argument values. An SQL-invoked procedure that is deterministic always returns the same 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, during invocation of that SQL-invoked routine, an SQL-implementation might, at two different times when the state of the SQL-data is the same, produce unequal results due to General Rules that specify implementation-dependent behavior.

An external routine either *does not possibly contain SQL* or *possibly contains SQL*. Only an external routine that possibly contains SQL may execute SQL-statements during its invocation.

An SQL-invoked routine may or may not *possibly read SQL-data*. Only an SQL-invoked routine that possibly reads SQL-data may read SQL-data during its invocation.

An SQL-invoked routine may or may not *possibly modify SQL-data*. 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.

When the <routine body> of an SQL-invoked routine is executed and the new SQL-session context for the SQL-session is created, the SQL-session user identifier in the new SQL-session context is set to the current user identifier in the SQL-session context that was active when the SQL-session caused the execution of the <routine body>. The authorization stack of this new SQL-session context is initially set to empty and a new pair of identifiers is immediately appended to the authorization stack such that:

- The user identifier is the newly initialized SQL-session user identifier.

- The role name is the current role name of the SQL-session context that was active when the SQL-session caused the execution of the <routine body>.

The identifiers in this new entry of the authorization stack are then modified depending on whether the SQL-invoked routine is an SQL routine or an external routine. If the SQL-invoked routine is an SQL routine, then, if the routine authorization identifier is a user identifier, the user identifier is set to the routine authorization identifier and the role name is set to null; otherwise, the role name is set to the routine authorization and the user identifier is set to null.

If the SQL-invoked routine is an external routine, then the identifiers are determined according to the external security characteristic of the SQL-invoked routine:

- If the external security characteristic is DEFINER, then:
 - If the routine authorization identifier is a user identifier, then the user identifier is set to the routine authorization identifier and the role name is set to the null value.
 - Otherwise, the role name is set to the routine authorization identifier and the user identifier is set to the null value.
- If the external security characteristic is INVOKER, then the identifiers remain unchanged.
- If the external security characteristic is IMPLEMENTATION DEFINED, then the identifiers are set to implementation-defined values.

An SQL-invoked routine that is an external routine also has an *external routine authorization identifier*, which is the <module authorization identifier>, if any, of the <SQL-client module definition> contained in the external program identified by the <routine body> of the external routine. If that <SQL-client module definition> does not specify a <module authorization identifier>, then the external routine authorization identifier is an implementation-defined authorization identifier.

The final value of the user identifier and role name in the authorization stack are used for privilege determination for access to the SQL objects, if any, referenced in the <SQL procedure statement>s that are executed during the execution of the <routine body>.

An SQL-invoked routine has a *routine SQL-path*, which is inherited from its containing SQL-schema, the current SQL-session, or the containing SQL-client module.

An SQL-invoked routine that is an external routine also has an *external routine SQL-path*, which is derived from the <module path specification>, if any, of the <SQL-client module definition> contained in the external program identified by the routine body of the external routine. If that <SQL-client module definition> does not specify a <module path specification>, then the external routine SQL-path is an implementation-defined SQL-path. For both SQL and external routines, the SQL-path of the current SQL-session is used to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is contained in a <preparable statement> 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> 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> or in a <direct SQL statement>.

An SQL-invoked routine is described by a *routine descriptor*. A routine descriptor contains:

- The routine name of the SQL-invoked routine.

- The <specific name> of the SQL-invoked routine.
- The routine authorization identifier of the SQL-invoked routine.
- The routine SQL-path of the SQL-invoked routine.
- The name of the language in which the body of the SQL-invoked routine is written.
- For each of the SQL-invoked routine's SQL parameters, the <SQL parameter name>, if it is specified, the <data type>, the ordinal position, and an indication of whether the SQL parameter is an input SQL parameter, an output SQL parameter, or both an input SQL parameter and an output SQL parameter.
- An indication of whether the SQL-invoked routine is an SQL-invoked function or an SQL-invoked procedure.
- If the SQL-invoked routine is an SQL-invoked procedure, then the maximum number of dynamic result sets.
- An indication of whether the SQL-invoked routine is deterministic or possibly non-deterministic.
- Indications of whether the SQL-invoked routine possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.
- If the SQL-invoked routine is an SQL-invoked function, then:
 - The <returns data type> of the SQL-invoked function.
 - If the <returns data type> simply contains <locator indication>, then an indication that the return value is a locator.
 - An indication of whether the SQL-invoked function is a type-preserving function or not.
 - An indication of whether the SQL-invoked function is a mutator function or not.
 - If the SQL-invoked function is a type-preserving function, then an indication of which parameter is the result parameter.
 - An indication of whether the SQL-invoked function is a null-call function.
- 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.
- If the SQL-invoked routine is an external routine, then:
 - The <external routine name> of the external routine.
 - The <parameter style> of the external routine.
 - If the external routine specifies a <result cast>, then an indication that it specifies a <result cast> and the <data type> specified in the <result cast>. If <result cast> contains <locator indication>, then an indication that the <data type> specified in the <result cast> has a locator indication.

- The external security characteristic of the external routine.
 - The external routine authorization identifier of the external routine.
 - The external routine SQL-path of the external routine.
 - The effective SQL parameter list of the external routine.
 - For every SQL parameter that has an associated from-sql function *FSF*, the specific name of *FSF*.
 - For every SQL parameter that has an associated to-sql function *TSF*, the specific name of *TSF*.
 - If the SQL-invoked routine is an external function and if it has a to-sql function *TRF* associated with the result, then the specific name of *TRF*.
 - For every SQL parameter whose <SQL parameter declaration> contains <locator indication>, an indication that the SQL parameter is a locator parameter.
- The <schema name> of the schema that includes the SQL-invoked routine.
 - If the SQL-invoked routine is an SQL-invoked function, then an indication of whether the SQL-invoked function is an SQL-invoked method.
 - 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.
 - If the SQL-invoked routine is an SQL-invoked method, then an indication of whether STATIC was specified.
 - An indication of whether the SQL-invoked routine is dependent on a user-defined type.

4.24 Built-in functions

Certain SQL operators whose invocation employs syntax similar to that of <routine invocation> are designated *built-in functions*. Those that are so designated are those that appear in the result of the following <query expression>:

```
SELECT DISTINCT ROUTINE_NAME
FROM INFORMATION_SCHEMA.ROUTINES
WHERE SPECIFIC_SCHEMA = 'INFORMATION_SCHEMA'
```

Some built-in functions are defined in this part of ISO/IEC 9075. It is implementation-defined whether there are built-in functions other than those defined in this part of ISO/IEC 9075.

4.25 SQL-paths

An SQL-path is a list of one or more <schema name>s that determines the search order for one of the following:

- The subject routine of a <routine invocation> whose <routine name> does not contain a <schema name>.

4.25 SQL-paths

— The user-defined type when the <user-defined type name> does not contain a <schema name>.

If the specification of an SQL-path does not specify a schema whose schema name is INFORMATION_SCHEMA, then INFORMATION_SCHEMA is assumed to precede all of the specified schema names.

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>.

4.26 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.

4.26.1 Status parameters

The SQLSTATE host parameter is a status parameter. It is set to status codes that indicate either that a call of the <externally-invoked procedure> completed successfully or that an exception condition was raised during execution of the <externally-invoked procedure>.

An <externally-invoked procedure> must specify the SQLSTATE host parameter. The SQLSTATE host parameter is a character string host parameter for which exception values are defined in Clause 22, "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 exception 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.

4.26.2 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.

4.26.3 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 character 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 bits, as appropriate) of the source value to indicate that truncation occurred and to indicate original length in characters or bits, as appropriate, of the source.

4.26.4 Locators

A host parameter, 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, an user-defined type locator, or an array 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 character large object.
- National character large object locator, a value of which identifies a national character large object.

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.

When the value at a site of binary large object type, character large object type, user-defined type or array 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.

4.26 Host parameters

A locator may be either *valid* or *invalid*. A host parameter 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 must 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.

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, a <rollback statement> that specifies a <savepoint clause> 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.

4.27 Diagnostics area

The diagnostics area is a place where completion and exception condition information is stored when an SQL-statement is executed. There is one diagnostics area associated with an SQL-agent, regardless of the number of SQL-client modules that the SQL-agent includes or the number of connections in use.

At the beginning of the execution of any <externally-invoked procedure> that contains an <SQL procedure statement> that is not an <SQL diagnostics statement>, the diagnostics area is emptied. An implementation must place information about a completion condition or an exception condition reported by SQLSTATE into this area. If other conditions are raised, an implementation may place information about them into this area.

<externally-invoked procedure>s containing <SQL diagnostics statement>s return a code indicating completion or exception conditions for that statement via SQLSTATE, but do not modify the diagnostics area.

An SQL-agent may choose the size of the diagnostics area with the <set transaction statement>; if an SQL-agent does not specify the size of the diagnostics area, then the size of the diagnostics area is implementation-dependent, but shall always be able to hold information about at least one condition. An SQL-implementation may place information into this area about fewer conditions than are specified. 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 specified by the SQLSTATE value.

4.28 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”.

4.29 Cursors

A cursor is specified by a <declare cursor>.

For every <declare cursor> in an SQL-client module, a cursor is effectively created when an SQL-transaction (see Subclause 4.32, "SQL-transactions") referencing the SQL-client module is initiated.

One of the properties that may be specified for a cursor determines whether or not it is a *holdable cursor*:

- A cursor that is not a holdable cursor is closed when the SQL-transaction in which it was created is terminated.
- A holdable cursor is not closed if that cursor is in the open state at the time that the SQL-transaction is terminated with a commit operation. A holdable cursor that is in the closed state at the time that the SQL-transaction is terminated remains closed. A holdable cursor is closed no matter what its state if the SQL-transaction is terminated with a rollback operation.
- A holdable cursor is closed and destroyed when the SQL-session in which it was created is terminated.

NOTE 31 – A holdable cursor may be said to be "holdable" or "held".

A cursor is in either the open state or the closed state. The initial state of a cursor is the closed state. A cursor is placed in the open state by an <open statement> and returned to the closed state by a <close statement> or a <rollback statement>. An open cursor that was not defined as a holdable cursor is also closed by a <commit statement>.

A cursor in the open state identifies a table, an ordering of the rows of that table, and a position relative to that ordering. If the <declare cursor> does not contain an <order by clause>, or contains an <order by clause> that does not specify the order of the rows completely, then the rows of the table have an order that is defined only to the extent that the <order by clause> specifies an order and is otherwise implementation-dependent.

When the ordering of a cursor is not defined by an <order by clause>, the relative position of two rows is implementation-dependent. When the ordering of a cursor is partially determined by an <order by clause>, then the relative positions of two rows are determined only by the <order by clause>; if the two rows have equal values for the purpose of evaluating the <order by clause>, then their relative positions are implementation-dependent.

A cursor is either *updatable* or *not updatable*. If the table identified by a cursor is not updatable or if *INSENSITIVE* is specified for the cursor, then the cursor is *not updatable*; otherwise, the cursor is updatable. The operations of update and delete are permitted for updatable cursors, subject to constraining Access Rules.

The position of a cursor in the open state is either before a certain row, on a certain row, or after the last row. If a cursor is on a row, then that row is the current row of the cursor. A cursor may be before the first row or after the last row of a table even though the table is empty. When a cursor is initially opened, the position of the cursor is before the first row.

A holdable cursor that has been held open retains its position when the new SQL-transaction is initiated. However, before either an <update statement: positioned> or a <delete statement: positioned> is permitted to reference that cursor in the new SQL-transaction, a <fetch statement> must be issued against the cursor.

4.29 Cursors

A <fetch statement> positions an open cursor on a specified row of the cursor's ordering and retrieves the values of the columns of that row. An <update statement: positioned> updates the current row of the cursor. A <delete statement: positioned> deletes the current row of the cursor.

If an error occurs during the execution of an SQL-statement that identifies a cursor, then, except where otherwise explicitly defined, the effect, if any, on the position or state of that cursor is implementation-dependent.

If a completion condition is raised during the execution of an SQL-statement that identifies a cursor, then the particular SQL-statement identifying that open cursor on which the completion condition is returned is implementation-dependent.

Another property of a cursor is its *sensitivity*, which may be sensitive, insensitive, or asensitive, depending on whether SENSITIVE, INSENSITIVE, or ASENSITIVE is specified or implied. The following paragraphs define several terms used to discuss issues relating to cursor sensitivity:

A change to SQL-data is said to be *independent* of a cursor *CR* if and only if it is not made by an <update statement: positioned> or a <delete statement: positioned> that is positioned on *CR*.

A change to SQL-data is said to be *significant* to *CR* if and only if it is independent of *CR*, and, had it been committed before *CR* was opened, would have caused the table associated with the cursor to be different in any respect.

A change to SQL-data is said to be *visible* to *CR* if and only if it has an effect on *CR* by inserting a row in *CR*, deleting a row from *CR*, changing the value of a column of a row of *CR*, or reordering the rows of *CR*.

If a cursor is open, and the SQL-transaction in which the cursor was opened makes a significant change to SQL-data, then whether that change is visible through that cursor before it is closed is determined as follows:

- If the cursor is insensitive, then significant changes are not visible.
- If the cursor is sensitive, then significant changes are visible.
- If the cursor is asensitive, then the visibility of significant changes is implementation-dependent.

If a holdable cursor is open during an SQL-transaction *T* and it is held open for a subsequent SQL-transaction, then whether any significant changes made to SQL-data (by *T* or any subsequent SQL-transaction in which the cursor is held open) are visible through that cursor in the subsequent SQL-transaction before that cursor is closed is determined as follows:

- If the cursor is insensitive, then significant changes are not visible.
- If the cursor is sensitive, then the visibility of significant changes is implementation-defined.
- If the cursor is asensitive, then the visibility of significant changes is implementation-dependent.

A <declare cursor> *DC* that specifies WITH RETURN is called a *result set cursor*. The <cursor specification> *CR* contained in *DC* defines a table *T*; the term *result set* is used to refer to *T*. A result set cursor, if declared in an SQL-invoked procedure and not closed when the procedure returns to its invoker, returns a result set to the invoker.

4.30 SQL-statements

4.30.1 Classes of SQL-statements

An SQL-statement is a string of characters that conforms to the Format and Syntax Rules specified in the parts of ISO/IEC 9075. Most SQL-statements can be prepared for execution and executed in an SQL-client module, in which case they are prepared when the SQL-client module is created and executed when the containing externally-invoked procedure is called (see Subclause 4.21, "SQL-client modules").

In this part of ISO/IEC 9075, there are at least two 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 must, be executed when no SQL-transaction is active.

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 entirely implementation-defined.

4.30.2 SQL-statements classified by function

The following are the main classes of SQL-statements:

- 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 a session is terminated.
- SQL-control statements.
- SQL-connection statements.
- SQL-session statements.
- SQL-diagnostics statements.

The following are the SQL-schema statements:

- <schema definition>
- <drop schema statement>
- <domain definition>
- <drop domain statement>
- <table definition>
- <drop table statement>

4.30 SQL-statements

- <view definition>
- <drop view statement>
- <assertion definition>
- <drop assertion statement>
- <alter table statement>
- <alter domain statement>
- <grant statement>
- <revoke statement>
- <character set definition>
- <drop character set statement>
- <collation definition>
- <drop collation statement>
- <translation definition>
- <drop translation 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>
- <transform definition>
- <drop transform statement>
- <schema routine>
- <alter routine statement>
- <drop routine statement>
- <role definition>
- <grant role statement>
- <drop role statement>

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>
- All 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>

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>

The following are the SQL-connection statements:

- <connect statement>
- <set connection statement>
- <disconnect statement>

The following are the SQL-control statements:

- <call statement>

4.30 SQL-statements

— <return statement>

The following are the SQL-session statements:

— <set session characteristics statement>

— <set session user identifier statement>

— <set role statement>

— <set local time zone statement>

The following are the SQL-diagnostics statements:

— <get diagnostics statement>

4.30.3 SQL-statements and transaction states

The following SQL-statements are transaction-initiating SQL-statements, *i.e.*, if there is no current SQL-transaction, and a statement of this class is executed, an SQL-transaction is initiated:

— All SQL-schema statements

— The SQL-transaction statements <commit statement> and <rollback statement>, if they specify AND CHAIN.

— The following SQL-data statements:

- <open statement>
- <close statement>
- <fetch statement>
- <select statement: single row>
- <insert statement>
- <delete statement: searched>
- <delete statement: positioned>
- <update statement: searched>
- <update statement: positioned>
- <free locator statement>
- <hold locator statement>

— <start transaction statement>

The following SQL-statements are not transaction-initiating SQL-statements, *i.e.*, if there is no current SQL-transaction, and a statement of this class is executed, no SQL-transaction is initiated.

— All SQL-transaction statements except <start transaction statement>s and <commit statement>s and <rollback statement>s that specify AND CHAIN.

- All SQL-connection statements
- All SQL-session statements
- All SQL-diagnostics statements
- The following SQL-data statements:
 - <temporary table declaration>
 - <declare cursor>

The following SQL-statements are possibly transaction-initiating SQL-statements:

- <return statement>

If the initiation of an SQL-transaction occurs in an atomic execution context, and an SQL-transaction has already completed in this context, then an exception condition is raised: *invalid transaction initiation*.

If an <SQL control statement> causes the evaluation of a <subquery> and there is no current SQL-transaction, then an SQL-transaction is initiated before evaluation of the <subquery>.

4.30.4 SQL-statement atomicity

The execution of all SQL-statements other than SQL-control statements is atomic with respect to recovery. Such an SQL-statement is called an *atomic SQL-statement*.

An *atomic execution context* is said to be active during the execution of an atomic SQL-statement or evaluation of a <subquery>. Within one atomic execution context, another atomic execution context may become active. This latter atomic execution context is said to be a *more recent atomic execution context*. During the execution of any SQL-statement S, if there is an atomic execution context for which no other atomic execution context is more recent, then it 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.

4.31 Basic security model

4.31.1 Authorization identifiers

An <authorization identifier> identifies a set of privileges. An <authorization identifier> can be either a <user identifier> or a <role name>. A <user identifier> represent a user of the database system. The mapping of <user identifier>s to operating system users is implementation-dependent. A <role name> represents a role.

4.31.1.1 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.

The context of an SQL-session contains a time-varying sequence of pairs of identifiers, known as the *authorization stack*, each pair consisting of a user identifier and a role name. This sequence of pairs is maintained using a “last-in, first-out” discipline, and effectively only the latest pair is visible. Initially, there is one pair whose user identifier is a copy of the SQL-session user identifier and whose role name is the null value. An additional pair is created by execution of an <externally invoked procedure> *EIP*. If the <module authorization clause> of the <SQL-client module definition> containing *EIP* specifies a <module authorization identifier> *MAI*, then *MAI* determines the value of the new pair, as follows. If *MAI* is a <user identifier>, then the user identifier of the pair is *MAI* and the role name is the null value; otherwise, the user identifier is the null value and the role name is *MAI*. If there is no <module authorization clause>, then the new pair is a copy of the previous last pair in the authorization stack. The pair placed in the stack during the execution of *EIP* is removed when that execution of *EIP* completes.

The latest pair in the authorization stack of the current SQL-session context determine the privileges for the execution of each SQL-statement. The user identifier in this pair is known as the current user identifier; the role name is known as the current role name. They may be determined using CURRENT_USER and CURRENT_ROLE, respectively. The current user identifier and the current role name are called the *current authorization identifiers*.

At a given time, the value of the current user identifier or the current role name can be a null value, but their values cannot both be null values at the same time. That is, there must always be a non-null current user identifier or current role name in order to have privileges to execute some SQL-statement (remember that 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 SQL-session user identifier. The <set role statement> changes the value of the current role name for the current SQL-session.

The phrase *current authorization identifier* refers to the situation in which the value of either the current user identifier or the current role name is a null value (and thus the value of the other <authorization identifier> is not a null value); the phrase is said to refer to the value of the current authorization identifier.

Let *A* be an <authorization identifier>. The phrase “the current authorization identifier is set to *A*” refers to:

- If *A* is a <user identifier>, then the current user identifier is set to *A* and the current role name is set to the null value.
- If *A* is a <role name>, then the current role name is set to *A* and the current user identifier is set to the null value.

4.31.1.2 SQL-client module authorization identifiers

An SQL-client module may specify an <authorization identifier>, called a <module authorization identifier>.

If a <module authorization identifier> is specified, then it is used as the current authorization identifier for the execution of all <externally-invoked procedure>s in the SQL-client module. If the <module authorization identifier> is not specified, 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 the SQL-client module.

4.31.1.3 SQL-schema authorization identifiers

A <schema definition> may specify an <authorization identifier>, called a <schema authorization identifier>, that represents the owner of that schema.

If the <schema authorization identifier> is specified, then it is used as the current authorization identifier for the creation of the schema. If the <schema authorization identifier> is not specified, then the <authorization identifier> specified by the <module authorization identifier> or the current user identifier of the SQL-session is used to determine the current authorization identifier for the creation of the schema.

4.31.2 Privileges

A privilege authorizes a given category of <action> to be performed on a specified base table, view, column, domain, character set, collation, translation, user-defined type, trigger, or SQL-invoked routine by a specified <authorization identifier>.

Each privilege is represented by a *privilege descriptor*. A privilege descriptor contains:

- The identification of the base table, view, column, domain, character set, collation, translation, user-defined type, table/method pair, trigger, or SQL-invoked routine module that the descriptor describes.
- The <authorization identifier> of the grantor of the privilege.
- The <authorization identifier> of the grantee of the privilege.
- Identification of the <action> that the privilege allows.
- An indication of whether or not the privilege is grantable.
- An indication of whether or not the privilege has the WITH HIERARCHY OPTION specified.

The <action>s that can be specified are:

- INSERT
- INSERT (<column name list>)
- UPDATE
- UPDATE (<column name list>)
- DELETE

- SELECT
- SELECT (<column name list>)
- SELECT (<privilege method list>)
- REFERENCES
- REFERENCES (<column name list>)
- USAGE
- UNDER
- TRIGGER
- EXECUTE

A privilege descriptor with an <action> of INSERT, UPDATE, DELETE, SELECT, TRIGGER, or REFERENCES is called a *table privilege descriptor* and identifies the existence of a privilege on the table identified by the privilege descriptor.

A privilege descriptor with an <action> of SELECT (<column name list>), INSERT (<column name list>), UPDATE (<column name list>), or REFERENCES (<column name list>) is called a *column privilege descriptor* and identifies the existence of a privilege on the columns in the table identified by the privilege descriptor.

A privilege descriptor with an <action> of SELECT (<privilege method list>) is called a *table/method privilege descriptor* and identifies the existence of a privilege on the methods of the structured type of the table identified by the privilege descriptor.

A table privilege descriptor specifies that the privilege identified by the <action> (unless the <action> is DELETE) is to be automatically granted by the grantor to the grantee on all columns subsequently added to the table.

A privilege descriptor with an <action> of USAGE is called a *usage privilege descriptor* and identifies the existence of a privilege on the domain, user-defined type, character set, collation, or translation identified by the privilege descriptor.

A privilege descriptor with an <action> of UNDER is called an *under privilege descriptor* and identifies the existence of the privilege on the structured type identified by the privilege descriptor.

A privilege descriptor with an <action> of EXECUTE is called an *execute privilege descriptor* and identifies the existence of a privilege on the SQL-invoked routine identified by the privilege descriptor.

A grantable privilege is a privilege associated with a schema that may be granted by a <grant statement>. The WITH GRANT OPTION clause of a <grant statement> specifies whether the <authorization identifier> recipient of a privilege (acting as a grantor) may grant it to others.

Privilege descriptors that represent privileges for the owner of an object have a special grantor value, “_SYSTEM”. This value is reflected in the Information Schema for all privileges that apply to the owner of the object.

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.

4.31.3 Roles

A role, identified by a <role name>, is a set of privileges defined by the union of the privileges defined by the privilege descriptors whose grantee is that <role name> and the sets of privileges of the <role name>s defined by the role authorization descriptors whose grantee is the first <role name>. A role may be granted to <authorization identifier>s with a <grant role statement>. No cycles of role grants are allowed.

The WITH ADMIN OPTION clause of the <grant role statement> specifies whether the recipient of a role may grant it to others.

Each grant is represented and identified by a *role authorization descriptor*. A role authorization descriptor includes:

- The <role name> of the role.
- The <authorization identifier> of the grantor.
- The <authorization identifier> of the grantee.
- An indication of whether or not the role was granted with the WITH ADMIN OPTION and hence is grantable.

Because roles may be granted to other roles, a role is said to “contain” other roles. The set of roles *X* contained in any role *A* is defined as the set of roles identified by role authorization descriptors whose grantee is *A*, together with all other roles contained by roles in *X*.

4.31.4 Security model definitions

The set of *applicable roles* for an <authorization identifier> consists of all roles defined by the role authorization descriptors whose grantee is that <authorization identifier> or PUBLIC together with all other roles they contain.

The set of *user privileges* for a <user identifier> consists of all privileges defined by the privilege descriptors whose grantee is either that <user identifier> or PUBLIC.

The set of *role privileges* for a <role name> consists of all privileges defined by the privilege descriptors whose grantee is either that <role name>, PUBLIC, or one of the applicable roles of that <role name>.

The set of *applicable privileges* for an <authorization identifier> is defined as:

- If that <authorization identifier> is a <user identifier>, then the set of user privileges for that <authorization identifier>.
- If that <authorization identifier> is a <role name>, then the set of role privileges for that <authorization identifier>.

The phrase *enabled roles* refers to:

- If the value of the current role name of the current SQL-session is the null value, then the empty set.
- Otherwise, the set of roles defined by the current role name of the current SQL-session together with its applicable roles.

The phrase *enabled authorization identifiers* refers to the set of <authorization identifier>s defined by the enabled roles together with the current user identifier of the current SQL-session, if its value is not a null value.

The phrase *enabled privileges* refers to:

- If the value of the current role name of the current SQL-session is a null value, then the empty set.
- Otherwise, the set of privileges defined by the role privileges of the current role name of the current SQL-session.

The phrase *current user privileges* refers to:

- If the value of the current user identifier of the current SQL-session is a null value, the empty set.
- Otherwise, the set of privileges defined by the user privileges of the current user identifier of the current SQL-session.

The phrase *current privileges* refers to the set of privileges defined by the current user privileges together with those defined by the enabled privileges.

4.32 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.

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.

Each SQL-client module 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 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.

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. If a <rollback statement> references a savepoint, then all changes made to SQL-data or schema subsequent to the establishment of the savepoint are canceled, and the SQL-transaction is restored to its state as it was immediately following the execution of the <savepoint statement>. Savepoints are destroyed when an SQL-transaction is terminated, or when a <release savepoint

statement> is executed. Savepoints may be redefined within an SQL-transaction by executing a <savepoint statement> that refers to a savepoint previously defined in the same SQL-transaction.

It is implementation-defined whether or not, or how, a <rollback statement> that references a <savepoint specifier> affects the contents of the diagnostics area, the contents of SQL descriptor areas, and the status of prepared statements.

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 *diagnostics area limit*, which is a positive integer that specifies the maximum number of conditions that can be placed in the diagnostics area during execution of an SQL-statement in this SQL-transaction.

Closing the cursor causes an effective check of all table constraints and assertions, the effective execution of all referential actions, and the re-evaluation of all matching rows and unique matching rows.

SQL-transactions initiated by different SQL-agents that access the same SQL-data or schemas and overlap in time are *concurrent 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 access mode 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 diagnostics 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.

4.32 SQL-transactions

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 10, "SQL-transaction isolation levels and the three phenomena" specifies the phenomena that are possible and not possible for a given isolation level.

Table 10—SQL-transaction isolation levels and the three phenomena

Level	<i>P1</i>	<i>P2</i>	<i>P3</i>
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 32 – 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.

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 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 11.38, “<trigger definition>”, and in the General Rules for Subclause 11.49, “<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, and
- 3) the effects of references to time-varying system variables such as **CURRENT_DATE** and **CURRENT_USER**.

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 must 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 coördinate the orderly termination of the encompassing transaction. When an SQL-transaction is part of an encompassing transaction that is controlled by an agent other than an SQL-agent and a <rollback statement> is initiated implicitly by an SQL-implementation, then the SQL-implementation will interact with that other agent to terminate that encompassing transaction. The specification of the interface between such agents and the SQL-implementation is beyond the scope of this part of ISO/IEC 9075. However, it is important to note that the semantics of an SQL-transaction remain as defined in the following sense:

- When an agent that is different from the SQL-agent requests the SQL-implementation to rollback an SQL-transaction, the General Rules of Subclause 16.7, “<rollback statement>”, are performed.
- When such an agent requests the SQL-implementation to commit an SQL-transaction, the General Rules of Subclause 16.6, “<commit statement>”, are performed. To guarantee orderly termination of the encompassing transaction, this commit operation may be processed in several phases not visible to the application; not all the General Rules of Subclause 16.6, “<commit statement>”, need to be executed in a single phase.

However, even in such environments, the SQL-agent interacts directly with the SQL-server to set characteristics (such as *read-only* or *read-write*, isolation level, and constraints mode) that are specific to the SQL-transaction model.

It is implementation-defined whether SQL-transactions that affect more than one SQL-server are supported. If such SQL-transactions are supported, then the part of each SQL-transaction that affects a single SQL-server is called a *branch transaction* or a branch of the SQL-transaction. If such SQL-transactions are supported, then they generally have all the same characteristics (access mode, diagnostics 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.

4.32 SQL-transactions

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 must 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 must 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 must 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 must have an isolation level of SERIALIZABLE.

- The diagnostics area limit of a branch of an SQL-transaction is always the same as the diagnostics area limit of the SQL-transaction; SET LOCAL TRANSACTION shall not specify a diagnostics 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*).

4.33 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: *connection exception — 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. 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 are implementation-defined.

An SQL-implementation must 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>.

4.34 SQL-sessions

An *SQL-session* spans the execution of a sequence of consecutive SQL-statements invoked by a single user from a single SQL-agent.

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 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, or an array locator is assigned a value of binary large object type, character large object type, user-defined type, or array 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.

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.

4.34 SQL-sessions

An SQL-session has a <user identifier> that is initially set to an implementation-defined value when the SQL-session is started, unless the SQL-session is started as a result of successful execution of a <connect statement>, in which case the initial <user identifier> of the SQL-session is set to the value of the implicit or explicit <connection user name> contained in the <connect statement>.

An SQL-session has an original local time zone displacement and a default local time zone displacement, which are values of data type INTERVAL HOUR TO MINUTE. Both the original local time zone displacement and the default local time zone displacement are initially set to the same implementation-defined value. The default local time zone displacement can subsequently be changed by successful execution of a <set local time zone statement>. The original local time zone displacement cannot be changed. It is also possible to set the default local time zone displacement to equal the value of the original local time zone displacement.

An SQL-invoked routine is *active* as soon as an SQL-statement executed by an SQL-agent causes invocation of an SQL-invoked routine and ceases to be active when execution of that invocation is complete.

At any time during an SQL-session, *containing SQL* is said to be *permitted* or *not permitted*. Similarly, *reading SQL-data* is said to be *permitted* or *not permitted* and *modifying SQL-data* is said to be *permitted* or *not permitted*.

An SQL-session has *enduring characteristics*. The enduring characteristics of an SQL-session are initially the same as the default values for the corresponding SQL-session characteristics. The enduring characteristics are changed by successful execution of a <set session characteristics statement> that specifies one or more enduring characteristics. Enduring characteristics that are not specified in a <set session characteristics statement> are not changed in any way by the successful execution of that statement.

SQL-sessions have the following enduring characteristics:

— *enduring transaction characteristics*

Each of the enduring characteristics are affected by a corresponding alternative in the <session characteristic> appearing in the <session characteristic list> of a <set session characteristics statement>.

An SQL-session has context that is preserved when an SQL-session is made dormant and restored when the SQL-session is made current. This context comprises:

- The current SQL-session identifier.
- The current user identifier.
- The current role name.
- The identities of all instances of temporary tables.
- The original time zone.
- The current default time zone.
- The current constraint mode for each integrity constraint.
- The current transaction access mode.
- The cursor position of all open cursors.
- The current transaction isolation level.

- The current SQL diagnostics area and its contents, along with the current diagnostics size.
- 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*.

NOTE 33 – 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.

4.34.1 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 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 execution context is always *atomic*. A routine execution context is either atomic or non-atomic. Every trigger execution context is atomic. Statement execution contexts are described in Subclause 4.30.3, “SQL-statements and transaction states”. Trigger execution contexts are described in Subclause 4.35, “Triggers”.

A routine execution context consists of:

- An indication as to whether or not an SQL-invoked routine is active.
- An indication as to whether or not containing SQL is permitted.
- An indication as to whether or not reading SQL-data is permitted.
- An indication as to whether or not modifying SQL-data is permitted.
- 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>.

4.35 Triggers

4.35 Triggers

A trigger is defined by a <trigger definition>. A <trigger definition> specifies a trigger that is described by a trigger descriptor. A trigger descriptor includes:

- The name of the trigger.
- The name of the base table that is 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 values correlation name, new values correlation name, old values table alias, or new values table alias.
- 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.

A *BEFORE trigger* is a trigger whose trigger event specifies BEFORE. An *AFTER trigger* is a trigger whose trigger event specifies AFTER.

A statement-level trigger is one specified using FOR EACH STATEMENT, while a row-level trigger is one specified using FOR EACH ROW. A trigger cannot be both a statement-level trigger and a row-level 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.

4.35.1 Triggered actions

A schema might include one or more trigger descriptors, each of which includes the definition of 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 triggering INSERT, DELETE, or UPDATE statement, in the case of a statement-level trigger) before or after rows are inserted into a table, rows are deleted from a table, or one or more columns are updated in rows of a table. The execution of such a triggered action resulting from the insertion, deletion, or updating of a table may cause the triggering of further triggered actions.

The <triggered SQL statement> of a triggered action is effectively executed either immediately before or immediately after the trigger event, as determined by the specified trigger action time.

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.

4.35.2 Execution of triggers

During the execution of an SQL-statement, zero or more *trigger execution contexts* exist, no more than one of which is *active*. The execution of an SQL-data change statement S_i creates a new trigger execution context TEC_i and causes TEC_i to become active. TEC_i remains in existence until the completion of S_i . An SQL-data change statement S_j that is executed before the completion of S_i preserves TEC_i and creates a new trigger execution context TEC_j that becomes the active one and remains in existence until the completion of S_j . At the completion of S_j , TEC_j ceases to exist and TEC_i is restored as the active trigger execution context.

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 contains a *set of transitions*, a set of statement-level triggers *considered as executed* for SC , and a set of row-level triggers, each paired with the set of rows in SC for which it is considered as executed.

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 transition represents a modification of a row in the subject table for the trigger event.

An old transition table is a set of old transition variables, each of which is the value of the row before the SQL-update operation is executed. A (possible empty) old transition table exists if the trigger event is UPDATE or DELETE. A new transition table is a set of new transition variables, each of which is the value of the row after the SQL-update operation is executed. A (possible empty) new transition table exists if the trigger event is UPDATE or INSERT.

If both transition tables exist, they have the same cardinality.

A <triggered action> may refer to the old transition table only by explicitly specifying an <old values table alias> and to the new transition table only by explicitly specifying a <new values table alias>. A <triggered action> may refer to a correlation name associated with the old transition table only by explicitly specifying an <old values correlation name>; this correlation name identifies the old transition variable. A <triggered action> may refer to a correlation name associated with the new transition table only by explicitly specifying a <new values correlation name>; this correlation name identifies the new transition variable. The scope of <old values table alias>, <new values table alias>, <old values correlation name>, and <new values correlation name> is the <triggered action> of the <trigger definition> that specifies it.

When execution of an SQL-data change statement S_i causes a trigger execution context TEC_i to come into existence, the set of state changes SSC_i is empty. Let TD be some trigger descriptor and let TE , ST , and CL be the trigger event, subject table, and column list of TD , respectively. A state change SC_{ij} arises in SSC_i when TE occurs during the execution of S_i and SSC_i does not already contain a state change identified by TE , ST , and CL . A transition T_{ijk} is added to SC_{ij} when a row is inserted into, updated in, or deleted from ST during the execution of S_i or the checking of referential constraints according to the General Rules of Subclause 11.8, "<referential constraint definition>", Subclause 14.6, "<delete statement: positioned>", Subclause 14.7, "<delete statement: searched>", Subclause 14.8, "<insert statement>", Subclause 14.9, "<update statement: positioned>", and Subclause 14.10, "<update statement: searched>".

A consequence of the execution of an SQL-data change statement that causes at least one transition to arise in some state change is called an *SQL-update operation*.

The transitions T_{ijk} added by an SQL-update operation SUO to a state change in SSC_i are the set of affected rows of SUO .

4.35 Triggers

When a state change SC_{ij} arises in SSC_i , one or more triggers are *activated* by SC_{ij} . A trigger TR is activated by SC_{ij} if and only if the subject table of TR is the subject table of SC_{ij} , the trigger event of TR is the trigger event of SC_{ij} , and the set of column names listed in TCL is the equivalent to the set of column names listed in $SCCL_{ij}$.

Whenever an SQL-update operation creates a new state change SC_{ij} , the BEFORE triggers activated by SC_{ij} are executed before the execution of the SQL-update operation. After the completion of the SQL-data change statement S_i that created a trigger execution context TEC_i , the AFTER triggers activated by any state changes SC_{ij} in the set of state changes SSC_i of TEC_i are executed.

A <triggered action> contained in a BEFORE or AFTER trigger that specifies FOR EACH ROW can refer to columns of old transition variables and new transition variables. Such references can be specified as <column reference>s, which includes as a special case <target specification>s and <simple target specification>s that identify the new transition variable. When a new transition variable is specified in a <target specification> or <simple target specification>, the column of the new transition variable that is identified by the <column name> contained in the <target specification> or <simple target specification> is effectively replaced by the value assigned to it.

4.36 Client-server operation

When an SQL-agent is active, it is bound in some implementation-defined manner to a single SQL-client. That SQL-client processes the explicit or implicit <SQL connection statement> for the first call to an <externally-invoked procedure> by an SQL-agent. The SQL-client communicates with, either directly or possibly through other agents such as RDA, one or more SQL-servers. An SQL-session involves an SQL-agent, an SQL-client, and a single SQL-server.

SQL-client modules associated with the SQL-agent exist in the SQL-environment containing the SQL-client associated with the SQL-agent.

Called <externally-invoked procedure>s 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 are processed by the SQL-server.

A call by the SQL-agent to an <externally-invoked procedure> containing an <SQL diagnostics statement> fetches information from the diagnostics area 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 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.

5 Lexical elements

5.1 <SQL terminal character>

Function

Define the terminal symbols of the SQL language and the elements of strings.

Format

```

<SQL terminal character> ::=
    <SQL language character>

<SQL language character> ::=
    <simple Latin letter>
    | <digit>
    | <SQL special character>

<simple Latin letter> ::=
    <simple Latin upper case letter>
    | <simple Latin lower case letter>

<simple Latin upper case letter> ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
    | P | Q | R | S | T | U | V | W | X | Y | Z

<simple Latin lower case letter> ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
    | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<SQL special character> ::=
    <space>
    | <double quote>
    | <percent>
    | <ampersand>
    | <quote>
    | <left paren>
    | <right paren>
    | <asterisk>
    | <plus sign>
    | <comma>
    | <minus sign>
    | <period>
    | <solidus>
    | <colon>
    | <semicolon>
    | <less than operator>
    | <equals operator>
    | <greater than operator>
    | <question mark>
    | <left bracket>
    | <right bracket>

```

```
| <circumflex>  
| <underscore>  
| <vertical bar>  
| <left brace>  
| <right brace>
```

<space> ::= !! *See the Syntax Rules*

<double quote> ::= "

<percent> ::= %

<ampersand> ::= &

<quote> ::= '

<left paren> ::= (

<right paren> ::=)

<asterisk> ::= *

<plus sign> ::= +

<comma> ::= ,

<minus sign> ::= -

<period> ::= .

<solidus> ::= /

<colon> ::= :

<semicolon> ::= ;

<less than operator> ::= <

<equals operator> ::= =

<greater than operator> ::= >

<question mark> ::= ?

<left bracket or trigraph> ::=
 <left bracket>
 | <left bracket trigraph>

<right bracket or trigraph> ::=
 <right bracket>
 | <right bracket trigraph>

<left bracket> ::= [

<left bracket trigraph> ::= ??(

<right bracket> ::=]

<right bracket trigraph> ::= ??)

<circumflex> ::= ^

<underscore> ::= _

<vertical bar> ::= |

<left brace> ::= {

<right brace> ::= }

Syntax Rules

- 1) Every character set shall contain a <space> character that is equivalent to U+0020.

Access Rules

None.

General Rules

- 1) There is a one-to-one correspondence between the symbols contained in <simple Latin upper case letter> and the symbols contained in <simple Latin lower case letter> such that, for all *i*, the symbol defined as the *i*-th alternative for <simple Latin upper case letter> corresponds to the symbol defined as the *i*-th alternative for <simple Latin lower case letter>.

Conformance Rules

None.

5.2 <token> and <separator>

Function

Specify lexical units (tokens and separators) that participate in SQL language.

Format

```
<token> ::=
    <nondelimiter token>
  | <delimiter token>

<nondelimiter token> ::=
    <regular identifier>
  | <key word>
  | <unsigned numeric literal>
  | <national character string literal>
  | <bit string literal>
  | <hex string literal>
  | <large object length token>
  | <multiplier>

<regular identifier> ::= <identifier body>

<identifier body> ::=
    <identifier start> [ { <underscore> | <identifier part> }... ]

<identifier start> ::=
    <initial alphabetic character>
  | <ideographic character>

<identifier part> ::=
    <alphabetic character>
  | <ideographic character>
  | <decimal digit character>
  | <identifier combining character>
  | <underscore>
  | <alternate underscore>
  | <extender character>
  | <identifier ignorable character>
  | <connector character>

<initial alphabetic character> ::= !! See the Syntax Rules
<ideographic character> ::= !! See the Syntax Rules
<alphabetic character> ::= !! See the Syntax Rules
<decimal digit character> ::= !! See the Syntax Rules
<identifier combining character> ::= !! See the Syntax Rules
<alternate underscore> ::= !! See the Syntax Rules
<extender character> ::= !! See the Syntax Rules
<identifier ignorable character> ::= !! See the Syntax Rules
<connector character> ::= !! See the Syntax Rules
```

<large object length token> ::=
 <digit>...<multiplier>

<multiplier> ::=
 K
 | M
 | G

<delimited identifier> ::=
 <double quote> <delimited identifier body> <double quote>

<delimited identifier body> ::= <delimited identifier part>...

<delimited identifier part> ::=
 <nondoublequote character>
 | <doublequote symbol>

<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>

<not equals operator> ::= <>

<greater than or equals operator> ::= >=

<less than or equals operator> ::= <=

<concatenation operator> ::= ||

<right arrow> ::= ->

<double colon> ::= ::

<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>[<minus sign>...]

<bracketed comment> ::= !! (See the Syntax Rules)
 <bracketed comment introducer>
 <bracketed comment contents>
 <bracketed comment terminator>

<bracketed comment introducer> ::= /*

<bracketed comment terminator> ::= */

<bracketed comment contents> ::=
 [{ <comment character> | <separator> }...]

<comment character> ::=
 <nonquote character>
 | <quote>

<newline> ::= !! See the Syntax Rules

<key word> ::=
 <reserved word>
 | <non-reserved word>

<non-reserved word> ::=
 ABS | ADA | ASENSITIVE | ASSIGNMENT | ASYMMETRIC | ATOMIC | AVG

 | BETWEEN | BIT_LENGTH | BITVAR

 | C | CALLED | CARDINALITY | CATALOG_NAME | CHAIN | CHAR_LENGTH
 | CHARACTER_LENGTH | CHARACTER_SET_CATALOG | CHARACTER_SET_NAME
 | CHARACTER_SET_SCHEMA | CHECKED | CLASS_ORIGIN | COALESCE | COBOL
 | COLLATION_CATALOG | COLLATION_NAME | COLLATION_SCHEMA | COLUMN_NAME
 | COMMAND_FUNCTION | COMMAND_FUNCTION_CODE | COMMITTED | CONDITION_NUMBER
 | CONNECTION_NAME | CONSTRAINT_CATALOG | CONSTRAINT_NAME | CONSTRAINT_SCHEMA
 | CONTAINS | CONVERT | COUNT | CURSOR_NAME

 | DATETIME_INTERVAL_CODE | DATETIME_INTERVAL_PRECISION | DEFINED | DEFINER
 | DISPATCH | DYNAMIC_FUNCTION | DYNAMIC_FUNCTION_CODE

 | EXISTING | EXISTS | EXTRACT

 | FINAL | FORTRAN

 | G | GENERATED | GRANTED

 | HIERARCHY | HOLD

 | IMPLEMENTATION | INFIX | INSENSITIVE | INSTANCE | INSTANTIABLE | INVOKER

 | K | KEY_MEMBER | KEY_TYPE

 | LENGTH | LOWER

 | M | MAX | MIN | MESSAGE_LENGTH | MESSAGE_OCTET_LENGTH | MESSAGE_TEXT
 | METHOD | MOD | MORE | MUMPS

 | NAME | NULLABLE | NUMBER | NULLIF

| OCTET_LENGTH | OPTIONS | OVERLAPS | OVERLAY | OVERRIDING

 | PASCAL | PARAMETER_MODE | PARAMETER_NAME | PARAMETER_ORDINAL_POSITION
 | PARAMETER_SPECIFIC_CATALOG | PARAMETER_SPECIFIC_NAME
 | PARAMETER_SPECIFIC_SCHEMA | PLI | POSITION

 | REPEATABLE | RETURNED_LENGTH | RETURNED_OCTET_LENGTH | RETURNED_SQLSTATE
 | ROUTINE_CATALOG | ROUTINE_NAME | ROUTINE_SCHEMA | ROW_COUNT

 | SCALE | SCHEMA_NAME | SECURITY | SELF | SENSITIVE | SERIALIZABLE | SERVER_NAME
 | SIMPLE | SOURCE | SPECIFIC_NAME | SIMILAR | SUBLIST | SUBSTRING | SUM | STYLE
 | SUBCLASS_ORIGIN | SYMMETRIC | SYSTEM

 | TABLE_NAME | TRANSACTIONS_COMMITTED | TRANSACTIONS_ROLLED_BACK
 | TRANSACTION_ACTIVE | TRANSFORM | TRANSFORMS | TRANSLATE | TRIGGER_CATALOG
 | TRIGGER_SCHEMA | TRIGGER_NAME | TRIM | TYPE

 | UNCOMMITTED | UNNAMED | UPPER | USER_DEFINED_TYPE_CATALOG
 | USER_DEFINED_TYPE_NAME | USER_DEFINED_TYPE_SCHEMA

 <reserved word> ::=
 ABSOLUTE | ACTION | ADD | ADMIN | AFTER | AGGREGATE
 | ALIAS | ALL | ALLOCATE | ALTER | AND | ANY | ARE | ARRAY | AS | ASC
 | ASSERTION | AT | AUTHORIZATION

 | BEFORE | BEGIN | BINARY | BIT | BLOB | BOOLEAN | BOTH | BREADTH | BY

 | CALL | CASCADE | CASCADED | CASE | CAST | CATALOG | CHAR | CHARACTER
 | CHECK | CLASS | CLOB | CLOSE | COLLATE | COLLATION | COLUMN | COMMIT
 | COMPLETION | CONNECT | CONNECTION | CONSTRAINT | CONSTRAINTS
 | CONSTRUCTOR | CONTINUE | CORRESPONDING | CREATE | CROSS | CUBE | CURRENT
 | CURRENT_DATE | CURRENT_PATH | CURRENT_ROLE | CURRENT_TIME | CURRENT_TIMESTAMP
 | CURRENT_USER | CURSOR | CYCLE

 | DATA | DATE | DAY | DEALLOCATE | DEC | DECIMAL | DECLARE | DEFAULT
 | DEFERRABLE | DEFERRED | DELETE | DEPTH | Deref | DESC | DESCRIBE | DESCRIPTOR
 | DESTROY | DESTRUCTOR | DETERMINISTIC | DICTIONARY | DIAGNOSTICS | DISCONNECT
 | DISTINCT | DOMAIN | DOUBLE | DROP | DYNAMIC

 | EACH | ELSE | END | END-EXEC | EQUALS | ESCAPE | EVERY | EXCEPT
 | EXCEPTION | EXEC | EXECUTE | EXTERNAL

 | FALSE | FETCH | FIRST | FLOAT | FOR | FOREIGN | FOUND | FROM | FREE | FULL
 | FUNCTION

 | GENERAL | GET | GLOBAL | GO | GOTO | GRANT | GROUP | GROUPING

 | HAVING | HOST | HOUR

 | IDENTITY | IGNORE | IMMEDIATE | IN | INDICATOR | INITIALIZE | INITIALLY
 | INNER | INOUT | INPUT | INSERT | INT | INTEGER | INTERSECT | INTERVAL
 | INTO | IS | ISOLATION | ITERATE

 | JOIN

 | KEY

 | LANGUAGE | LARGE | LAST | LATERAL | LEADING | LEFT | LESS | LEVEL | LIKE
 | LIMIT
 | LOCAL | LOCALTIME | LOCALTIMESTAMP | LOCATOR

| MAP | MATCH | MINUTE | MODIFIES | MODIFY | MODULE | MONTH

| NAMES | NATIONAL | NATURAL | NCHAR | NCLOB | NEW | NEXT | NO | NONE
| NOT | NULL | NUMERIC

| OBJECT | OF | OFF | OLD | ON | ONLY | OPEN | OPERATION | OPTION
| OR | ORDER | ORDINALITY | OUT | OUTER | OUTPUT

| PAD | PARAMETER | PARAMETERS | PARTIAL | PATH | POSTFIX | PRECISION | PREFIX
| PREORDER | PREPARE | PRESERVE | PRIMARY
| PRIOR | PRIVILEGES | PROCEDURE | PUBLIC

| READ | READS | REAL | RECURSIVE | REF | REFERENCES | REFERENCING | RELATIVE
| RESTRICT | RESULT | RETURN | RETURNS | REVOKE | RIGHT
| ROLE | ROLLBACK | ROLLUP | ROUTINE | ROW | ROWS

| SAVEPOINT | SCHEMA | SCROLL | SCOPE | SEARCH | SECOND | SECTION | SELECT
| SEQUENCE | SESSION | SESSION_USER | SET | SETS | SIZE | SMALLINT | SOME | SPACE
| SPECIFIC | SPECIFICTYPE | SQL | SQLEXCEPTION | SQLSTATE | SQLWARNING | START
| STATE | STATEMENT | STATIC | STRUCTURE | SYSTEM_USER

| TABLE | TEMPORARY | TERMINATE | THAN | THEN | TIME | TIMESTAMP
| TIMEZONE_HOUR | TIMEZONE_MINUTE | TO | TRAILING | TRANSACTION | TRANSLATION
| TREAT | TRIGGER | TRUE

| UNDER | UNION | UNIQUE | UNKNOWN
| UNNEST | UPDATE | USAGE | USER | USING

| VALUE | VALUES | VARCHAR | VARIABLE | VARYING | VIEW

| WHEN | WHENEVER | WHERE | WITH | WITHOUT | WORK | WRITE

| YEAR

| ZONE

NOTE 34 – The list of <reserved word>s is considerably longer than the analogous list of <key word>s in ISO/IEC 9075:1992. To assist users of ISO/IEC 9075 avoid such words in a possible future revision, the following list of potential <reserved word>s is provided. Readers must understand that there is no guarantee that all of these words will, in fact, become <reserved word>s in any future revision; furthermore, it is almost certain that additional words will be added to this list as any possible future revision emerges.

Assurance that a <regular identifier> will not become a <reserved word> in a possible future revision of ISO/IEC 9075 can be obtained by including a <digit> or an <underscore> in the <regular identifier>, and by taking care to avoid identifiers beginning with CURRENT_, SESSION_, SYSTEM_, or TIMEZONE_, or ending in _LENGTH.

The words are:

No words are potentially reserved.

Syntax Rules

- 1) An <alphanumeric character> is any character with the Unicode alphabetic property.

NOTE 35 – Characters with the “alphabetic” property include characters that are called “letters” and characters that are called “syllables”.

- 2) An <initial alphabetic character> is an <alphabetic character> that does not have the Unicode “combining” property.
- 3) An <ideographic character> is a character with the Unicode “ideographic” property.
- 4) A <decimal digit character> is a character with the Unicode “decimaldigit” property.

- 5) An <identifier combining character> is a character with the Unicode “combining” property, except for U+06DD, U+06DE, U+20DD, U+20DE, U+20DF, and U+20E0.

NOTE 36 – U+06DD is “Arabic End of Ayah”, U+06DE is “Arabic Start of Rub El Hizb”, U+20DD is “Combining Enclosing Circle”; U+20DE is “Combining Enclosing Square”, U+20DF is “Combining Enclosing Diamond”, and U+20E0 is “Combining Enclosing Circle Backslash”.

- 6) An <extender character> is any of U+00B7, U+02D0, U+02D1, U+0640, U+0E46, U+0EC6, U+3005, U+3031 through U+3035 inclusive, U+309B through U+309E inclusive, U+30FC through U+30FE inclusive, U+FF70, U+FF9E, and U+FF9F.

NOTE 37 – U+00B7 is “Middle Dot”, U+02D0 is “Modifier Letter Triangular Colon”, U+02D1 is “Modifier Letter Half Triangular Colon”, U+0640 is “Arabic Tatweel”, U+0E46 is “Thai Character Maiyamok”, U+0EC6 is “Lao Ko La”, U+3005 is “Ideographic Iteration Mark”, U+3031 through U+3035 inclusive are variations of Vertical Kana Repeat Marks, U+309B through U+309E inclusive are variations of Combining Katakana-Hiragana Sound Marks and Hiragana Iteration Marks, U+30FC through U+30FE inclusive are variations of Katakana-Hiragana Prolonged Sound Mark and Katakana Iteration Marks, U+FF70 is “Halfwidth Katakana-Hiragana Prolonged Sound Mark”, U+FF9E is “Halfwidth Katakana Voiced Sound Mark”, and U+FF9F is “Halfwidth Katakana Semi-voiced Sound Mark”.

- 7) An <identifier ignorable character> is any of U+200C through U+200F inclusive, U+202A through U+202E inclusive, U+206A through U+206F inclusive, and U+FEFF.

NOTE 38 – U+200C is “Zero Width Non-Joiner”, U+200D is “Zero Width Joiner”, U+200E is “Left-To-Right Mark”, U+200F is “Right-To-Left Mark”, U+202A is “Left-To-Right Embedding”, U+202B is “Right-To-Left Embedding”, U+202C is “Pop Directional Formatting”, U+202D is “Left-To-Right Override”, U+202E is “Right-To-Left Override”, U+206A is “Inhibit Symmetric Swapping”, U+206B is “Activate Symmetric Swapping”, U+206C is “Inhibit Arabic Form Shaping”, U+206D is “Activate Arabic Form Shaping”, U+206E is “National Digit Shapes”, U+206F is “Nominal Digit Shapes”, and U+FEFF is “Zero-Width No-Break Space”.

- 8) An <alternate underscore> is any of U+FE33, U+FE34, U+FE4D, U+FE4E, U+FE4F, and U+FF3F.

NOTE 39 – U+FE33 is “Presentation Form for Vertical Low Line”, U+FE34 is “Presentation Form for Vertical Wavy Low Line”, U+FE4D is “Dashed Low Line”, U+FE4E is “Centreline Low Line”, U+FE4F is “Wavy Low Line” and U+FF3F is “Fullwidth Low Line”.

- 9) A <connector character> is any of U+203F or U+2040.

NOTE 40 – U+203F is “Undertie” and U+2040 is “Character Tie”.

- 10) <white space> is any consecutive sequence of characters each of which satisfies the definition of white space found in Subclause 3.1.5, “Definitions provided in Part 2”.

- 11) <newline> is the implementation-defined end-of-line indicator.

NOTE 41 – <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.

- 12) With the exception of the <space> character explicitly contained in <timestamp string> and <interval string> and the permitted <separator>s in <bit string literal>s and <hex string literal>s, a <token>, other than a <character string literal>, a <national character string literal>, or a <delimited identifier>, shall not contain a <space> character or other <separator>.

13) A <nondoublequote character> is one of:

- a) Any <SQL language character> other than a <double quote>;
- b) Any character other than a <double quote> in the character repertoire identified by the <module character set specification>; or
- c) Any character other than a <double quote> in the character repertoire identified by the <character set specification>.

14) Any <token> may be followed by a <separator>. A <nondelimiter token> shall be followed by a <delimiter token> or a <separator>.

NOTE 42 – 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>.

15) There shall be no <space> nor <newline> separating the <minus sign>s of a <simple comment introducer>.

16) There shall be no <separator> separating any two <digit>s or separating a <digit> and <multiplier> of a <large object length token>.

17) 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 43 – 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>.

18) SQL text containing one or more instances of <comment> is equivalent to the same SQL text with the <comment> replaced with <newline>.

19) In a <regular identifier>, the number of <identifier part>s shall be less than 128.

20) The <delimited identifier body> of a <delimited identifier> shall not comprise more than 128 <delimited identifier part>s.

21) 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 translated 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*.

22) 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 44 – Any lower-case letters for which there are no upper-case equivalents are left in their lower-case form.

- 23) The <identifier body> of a <regular identifier> is equivalent to an <identifier body> in which every letter that is a lower-case letter is replaced by the equivalent upper-case letter or letters. This treatment includes determination of equivalence, representation in the Information and Definition Schemas, representation in the diagnostics area, and similar uses.
- 24) The <identifier body> of a <regular 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 45 – Assurance that a <regular identifier> will not become a <reserved word> in a possible future revision of ISO/IEC 9075 can be obtained by including a <digit> or an <underscore> in the <regular identifier>, and by taking care to avoid identifiers beginning with CURRENT_, SESSION_, SYSTEM_, or TIMEZONE_, or ending in _LENGTH.
- 25) Two <regular identifier>s are equivalent if their <identifier body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".
- 26) A <regular identifier> and a <delimited identifier> are equivalent if the <identifier body> of the <regular identifier> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters) 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 an implementation-defined collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".
- 27) 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>".
- 28) 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 <underscore>s plus 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 46 – 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>.

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>
  | <bit string literal>
  | <hex 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> }... ]

<bit string literal> ::=
    B <quote> [ <bit>... ] <quote>
    [ { <separator> <quote> [ <bit>... ] <quote> }... ]

<hex string literal> ::=
    X <quote> [ <hexit>... ] <quote>
    [ { <separator> <quote> [ <hexit>... ] <quote> }... ]

<binary string literal> ::=
    X <quote> [ { <hexit> <hexit> }... ] <quote>
    [ { <separator> <quote> [ { <hexit> <hexit> }... ] <quote> }... ]

<bit> ::= 0 | 1

```

5.3 <literal>

<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>
    | [ <years value> <minus sign> ] <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

```

5.3 <literal>

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 47 – 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 <bit string literal>, the sequence

<quote> <bit>... <quote> <separator> <quote> <bit>... <quote>

is equivalent to the sequence

<quote> <bit>... <bit>... <quote>

NOTE 48 – The <bit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <bit string literal>.

- 3) In a <hex string literal>, the sequence

<quote> <hexit>... <quote> <separator> <quote> <hexit>... <quote>

is equivalent to the sequence

<quote> <hexit>... <hexit>... <quote>

NOTE 49 – The <hexit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <hex string literal>.

- 4) 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 50 – The <hexits> in the equivalent sequence are in the same sequence and relative sequence as in the original <binary string literal>.

- 5) In a <character string literal>, <national character string literal>, <bit string literal>, <binary string literal>, or <hex string literal>, a <separator> shall contain a <newline>.

- 6) 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>.

- 7) A <nonquote character> is one of:

- Any <SQL language character> other than a <quote>;
- Any character other than a <quote> in the character repertoire identified by the <module character set specification>; or
- Any character other than a <quote> in the character repertoire identified by the <character set specification> or implied by “N”.

8) 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 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>.

- 9) 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 51 – <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).

- 10) 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>.

- 11) The declared type of a <bit string literal> is fixed-length bit string. The length of a <bit string literal> is the number of bits that it contains.

- 12) The declared type of a <hex string literal> is fixed-length bit string. Each <hexit> appearing in the literal is equivalent to a quartet of bits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are interpreted as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111, respectively. The <hexit>s a, b, c, d, e, and f have respectively the same values as the <hexit>s A, B, C, D, E, and F.

- 13) The declared type of a <binary string literal> is binary string. Each <hexit> appearing in the literal is equivalent to a quartet of bits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are interpreted as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111, respectively. The <hexit>s a, b, c, d, e, and f have respectively the same values as the <hexit>s A, B, C, D, E, and F.

- 14) An <exact numeric literal> without a <period> has an implied <period> following the last <digit>.

- 15) The declared type of an <exact numeric literal> is exact numeric. The precision of an <exact numeric literal> is the number of <digit>s that it contains. The scale of an <exact numeric literal> is the number of <digit>s to the right of the <period>.

- 16) The declared type of an <approximate numeric literal> is approximate numeric. The precision of an <approximate numeric literal> is the precision of its <mantissa>.

- 17) The declared type of a <date literal> is DATE.

5.3 <literal>

- 18) 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.
- 19) 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.
- 20) 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.
- 21) Let *datetime component* be either <years value>, <months value>, <days value>, <hours value>, <minutes value>, or <seconds value>.
- 22) 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>.
- 23) 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.
- 24) 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.
- 25) Within the definition of an <interval literal>, the <datetime value>s are constrained by the natural rules for intervals according to the Gregorian calendar.
- 26) Within the definition of a <year-month literal>, the <interval qualifier> shall not specify DAY, HOUR, MINUTE, or SECOND. Within the definition of a <day-time literal>, the <interval qualifier> shall not specify YEAR or MONTH.
- 27) Within the definition of a <datetime literal>, the value of the <time zone interval> shall be in the range −12:59 to +13:00.

Access Rules

None.

General Rules

- 1) The value of a <character string literal> is the sequence of <character representation>s that it contains.
- 2) The value of a <bit string literal> or a <hex string literal> is the sequence of bits that it contains.

- 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 .

5.3 <literal>

b) Otherwise, the value of the <datetime literal> is *DV*.

NOTE 52 – 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 default SQL-session time zone displacement 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 53 – The null value of the boolean data type is equivalent to the unknown truth value (see Subclause 4.6, “Boolean types”).

Conformance Rules

- 1) Without Feature T031, “BOOLEAN data type”, a <general literal> shall not be a <boolean literal>.
- 2) Without Feature F555, “Enhanced seconds precision”, an <unsigned integer> that is a <seconds fraction> that is contained in a <timestamp literal> shall not contain more than 6 <digit>s. A <time literal> shall not contain a <seconds fraction>.
- 3) Without Feature F511, “BIT data type”, a <general literal> shall not be a <bit string literal> or a <hex string literal>.
- 4) Without Feature F421, “National character”, a <general literal> shall not be a <national character string literal>.
- 5) Without Feature F052, “Intervals and datetime arithmetic”, a <general literal> shall not be an <interval literal>.
- 6) Without Feature F271, “Compound character literals”, conforming SQL language shall contain exactly one repetition of <character representation> (that is, it shall contain exactly one sequence of “<quote> <character representation>... <quote>”).
- 7) Without Feature F451, “Character set definition”, or Feature F461, “Named character sets”, a <character string literal> shall not specify a <character set specification>.
- 8) Without Feature F411, “Time zone specification”, conforming Core SQL shall not specify a <time zone interval>.
- 9) Without Feature T041, “Basic LOB data type support”, conforming Core SQL language shall not contain any <binary string literal>.

5.4 Names and identifiers

Function

Specify names.

Format

<identifier> ::=
 <actual identifier>

 <actual identifier> ::=
 <regular identifier>
 | <delimited identifier>

 <SQL language identifier> ::=
 <SQL language identifier start>
 [{ <underscore> | <SQL language identifier part> }...]

 <SQL language identifier start> ::= <simple Latin letter>

 <SQL language identifier part> ::=
 <simple Latin letter>
 | <digit>

 <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>
 | MODULE

 <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>

<translation name> ::= <schema qualified name>

<form-of-use conversion name> ::= <schema qualified name>

<user-defined type name> ::= <schema qualified type name>

<schema qualified type name> ::=
 [<schema name> <period>] <qualified identifier>

<attribute name> ::=
 <identifier>

<field name> ::= <identifier>

<savepoint name> ::= <identifier>

<role name> ::= <identifier>

<user identifier> ::= <identifier>

<connection name> ::= <simple value specification>

<SQL-server name> ::= <simple value specification>

<connection user name> ::= <simple value specification>

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 equivalent upper-case letter or letters. This treatment includes determination of equivalence, representation in the Information and Definition Schemas, representation in the diagnostics area, and similar uses.
- 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 54 – 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 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 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 *M* and the <module contents> of *M* shall contain a <temporary table declaration> *TT* whose <table name> has a <qualified identifier> equal 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> equal to *QI*.
- 6) If a <cursor name> *CN* with a <qualified identifier> *QI* does not contain a <local qualifier>, then the <local qualifier> MODULE is implicit.
- 7) Let *CN* be a <cursor name> with a <qualified identifier> *QI* and a <local qualifier> *LQ*. *LQ* shall be "MODULE" and *CN* shall be contained in an SQL-client module whose <module contents> contain a <declare cursor> whose <cursor name> is *CN*.
- 8) Case:
 - a) If <user-defined type name> *UDTN* with a <qualified identifier> *QI* is simply contained in <user-defined type>, 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 in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.
- B) 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) ≤ *i* ≤ *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) ≤ *i* ≤ *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) Otherwise,

Case:

- i) 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>.
 - ii) Otherwise, the implicit <schema name> of *UDTN* is the <schema name> that is specified or implicit in <SQL-client module definition>.
- 9) Two <schema qualified 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 <schema qualified name> does not contain a <schema name>, then

Case:

- a) If the <schema qualified name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.
- b) Otherwise, the <schema name> that is specified or implicit for the <SQL-client module definition> is implicit.

- 12) If a <schema name> does not contain a <catalog name>, then

Case:

- a) If the <unqualified schema name> is contained in a <module authorization clause>, then an implementation-defined <catalog name> is implicit.
- b) 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.
- c) 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, then the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.
 - ii) Otherwise, an implementation-defined <catalog name> is implicit.
 - d) Otherwise, the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.
- 13) Two <schema qualified name>s are equivalent if and only if they have the same <qualified identifier> and the same <schema name>, regardless of whether the <schema name>s are implicit or explicit.
- 14) Two <character set name>s are equivalent if and only if they have the same <SQL language identifier> and the same <schema name>, regardless of whether the <schema name>s are implicit or explicit.
- 15) Two <schema name>s are equivalent if and only if they have the same <unqualified schema name> and the same <catalog name>, regardless of whether the <catalog name>s are implicit or explicit.
- 16) An <identifier> that is a <correlation name> is associated with a table within a particular scope. The scope of a <correlation name> is either a <select statement: single row>, <subquery>, or <query specification> (see Subclause 7.6, "<table reference>"), or is a <trigger definition> (see Subclause 11.38, "<trigger definition>"). Scopes may be nested. In different scopes, the same <correlation name> may be associated with different tables or with the same table.
- 17) No <authorization identifier> shall specify "PUBLIC".
- 18) Those <identifier>s that are valid <authorization identifier>s are implementation-defined.
- 19) Those <identifier>s that are valid <catalog name>s are implementation-defined.
- 20) If a <character set name> does not specify a <schema name>, then INFORMATION_SCHEMA is implicit.
- 21) If a <collation name> does not specify a <schema name>, then INFORMATION_SCHEMA is implicit.
- 22) If a <translation name> does not specify a <schema name>, then INFORMATION_SCHEMA is implicit.

- 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) If a <form-of-use conversion name> does not specify a <schema name>, then INFORMATION_SCHEMA is implicit; otherwise, INFORMATION_SCHEMA shall be specified.

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 collating sequence.
- 20) A <character set name> identifies a character set.
- 21) A <translation name> identifies a character translation.

- 22) A <form-of-use conversion name> identifies a form-of-use conversion. All <form-of-use conversion 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 <field name> identifies a field.
- 28) A <role name> identifies a role.
- 29) A <user identifier> identifies a user.
- 30) If the <form-of-use conversion name> does not contain an explicit a <schema name>, then INFORMATION_SCHEMA is implicit; otherwise, INFORMATION_SCHEMA shall be specified.

Conformance Rules

- 1) Without Feature T271, "Savepoints", conforming SQL language shall not contain any <savepoint name>.
- 2) Without Feature T331, "Basic roles", conforming SQL language shall not contain any <role name>.
- 3) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", conforming SQL language shall not contain any <query name>.
- 4) Without Feature S023, "Basic structured types", conforming SQL language shall not contain any <attribute name>.
- 5) Without Feature T051, "Row types", conforming SQL language shall not contain any <field name>.
- 6) Without Feature F651, "Catalog name qualifiers", conforming SQL language shall not contain any explicit <catalog name>.
- 7) Without Feature F771, "Connection management", conforming SQL language shall not contain any explicit <connection name>.
- 8) Without Feature F691, "Collation and translation", conforming SQL language shall not contain any explicit <collation name>, <translation name>, or <form-of-use conversion name>.
- 9) Without Feature F821, "Local table references", conforming SQL language shall not specify MODULE in a <local or schema qualifier> or <local qualified name>.
- 10) Without Feature F251, "Domain support", conforming SQL language shall not contain any <domain name>.
- 11) Without Feature F491, "Constraint management", conforming SQL language shall not contain any <constraint name>.

- 12) Without Feature F451, “Character set definition”, or Feature F461, “Named character sets”, conforming SQL language shall not contain any <character set name>.
- 13) Without Feature T601, “Local cursor references”, a <cursor name> shall not specify <local qualifier>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

6 Scalar expressions

6.1 <data type>

Function

Specify a data type.

Format

```

<data type> ::=
    <predefined type>
  | <row type>
  | <user-defined type>
  | <reference type>
  | <collection type>

<predefined type> ::=
    <character string type> [ CHARACTER SET <character set specification> ]
  | <national character string type>
  | <binary large object string type>
  | <bit 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 [ <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 [ <left paren> <large object length> <right paren> ]
  | NCHAR LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | NCLOB [ <left paren> <large object length> <right paren> ]

<binary large object string type> ::=
    BINARY LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | BLOB [ <left paren> <large object length> <right paren> ]

<bit string type> ::=
    BIT [ <left paren> <length> <right paren> ]

```

6.1 <data type>

```

    | BIT VARYING <left paren> <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> ]
    | INTEGER
    | INT
    | SMALLINT

<approximate numeric type> ::=
    FLOAT [ <left paren> <precision> <right paren> ]
    | REAL
    | DOUBLE PRECISION

<length> ::= <unsigned integer>

<large object length> ::=
    <unsigned integer> [ <multiplier> ]
    | <large object length token>

<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> ::= <user-defined type>

<user-defined type> ::= <user-defined type name>

<collection type> ::=
    <data type> <array specification>

<array specification> ::=
    <collection type constructor>
    <left bracket or trigraph> <unsigned integer> <right bracket or trigraph>

<collection type constructor> ::=
    ARRAY

```

Syntax Rules

- 1) CHAR is equivalent to CHARACTER. DEC is equivalent to DECIMAL. INT is equivalent to INTEGER. VARCHAR is equivalent to CHARACTER VARYING. NCHAR is equivalent to NATIONAL CHARACTER. CLOB is equivalent to CHARACTER LARGE OBJECT. NCLOB is equivalent to NATIONAL CHARACTER LARGE OBJECT. BLOB is equivalent to BINARY LARGE OBJECT.
- 2) "NATIONAL CHARACTER" is equivalent to the corresponding <character string type> with a specification of "CHARACTER SET CSN", where "CSN" is an implementation-defined <character set name>.
- 3) The value of a <length> or a <precision> shall be greater than 0 (zero).
- 4) If <length> is omitted, then a <length> of 1 (one) is implicit.
- 5) If <large object length> is omitted, then an implementation-defined <large object length> is implicit.
- 6) The numeric value of a <large object length> is determined as follows.
Case:
 - a) If <large object length> immediately contains <unsigned integer> and does not immediately contain <multiplier>, then the numeric value of <large object length> is the numeric value of the specified <unsigned integer>.
 - b) If <large object length> immediately contains <large object length token> or immediately contains <unsigned integer> and <multiplier>, then let *D* be the value of the specified <unsigned integer> or the numeric value of the sequence of <digit>s of <large object length token> interpreted as an <unsigned integer>. The numeric value of <large object length> is the numeric value resulting from the multiplication of *D* and *MS*, then *MS* is:
 - i) If <multiplier> is K, then 1,024.
 - ii) If <multiplier> is M, then 1,048,576.
 - iii) If <multiplier> is G, then 1,073,741,824.

6.1 <data type>

- 7) If a <scale> is omitted, then a <scale> of 0 (zero) is implicit.
- 8) If a <precision> is omitted, then an implementation-defined <precision> is implicit.
- 9) CHARACTER specifies the data type character string.
- 10) Characters in a character string are numbered beginning with 1 (one).
- 11) 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>.

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.
- 12) 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 55 – Subclause 11.23, “<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.
- 13) The character set named SQL_TEXT is an implementation-defined character set whose character repertoire is SQL_TEXT.

NOTE 56 – The character repertoire SQL_TEXT is defined in Subclause 4.2, “Character strings”.
- 14) The character set named SQL_IDENTIFIER is an implementation-defined character set whose character repertoire is SQL_IDENTIFIER.

NOTE 57 – The character repertoire SQL_IDENTIFIER is defined in Subclause 4.2, “Character strings”.
- 15) BINARY LARGE OBJECT specifies the data type binary string.
- 16) Octets in a binary large object string are numbered beginning with 1 (one). The length in octets of the string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.
- 17) BIT specifies the data type bit string.
- 18) Bits in a bit string are numbered beginning with 1 (one).
- 19) Case:
 - a) If VARYING is not specified in <bit string type>, then the length in bits of the bit string is fixed and is the value of <length>.

- b) If VARYING is specified in <bit string type>, then the length in bits of the string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <length>.

The maximum value of <length> is implementation-defined. <length> shall not be greater than this maximum value.

- 20) The <scale> of an <exact numeric type> shall not be greater than the <precision> of the <exact numeric type>.
- 21) 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.
- 22) NUMERIC specifies the data type exact numeric, with the decimal precision and scale specified by the <precision> and <scale>.
- 23) 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>.
- 24) INTEGER specifies the data type exact numeric, with binary or decimal precision and scale of 0 (zero). The choice of binary versus decimal precision is implementation-defined, but shall be the same as SMALLINT.
- 25) SMALLINT specifies 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 shall be the same as INTEGER. The precision of SMALLINT shall be less 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.

6.1 <data type>

- 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 58 – 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) Within the non-null values of a <datetime type>, the value of the time zone interval shall be in the range –12:59 to +13:00.
- NOTE 59 – 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.
- 37) If <data type> is a <reference type>, then there shall exist 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.
- 38) The <table name> contained in a <scope clause> shall identify a referenceable table whose structured type is UDTN.
- 39) 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.
- 40) If <collection type> CT immediately contains an <array specification>, then the <data type> immediately contained in CT shall not contain a <collection type>.
- 41) A <collection type> CT that immediately contains an <array specification> specifies an *array type*. The <data type> immediately contained in CT is the *element type* of the array type. The <unsigned integer> immediately contained in <array specification> is the *maximum cardinality* of a site of data type CT.
- 42) <row type> specifies the row data type.
- 43) BOOLEAN specifies the boolean data type.

Access Rules

- 1) If <reference type> is specified, then

Case:

 - a) If <reference type> is contained in an <SQL schema statement>, then the applicable privileges shall include the USAGE privilege on the user-defined type identified by the <user-defined type name> simply contained in <referenced type>.

- b) Otherwise, the current privileges shall include the USAGE privilege on the user-defined type identified by the <user-defined type name> simply contained in <referenced type>.

NOTE 60 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) If any specification or operation attempts to cause an item of a character type to contain a character that is not a member of the character repertoire associated with the character item, then an exception condition is raised: *data exception — character not in repertoire*.

- 2) 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 61 – A <datetime type> contains no other fields than those specified by the preceding Rule.

- 3) 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>.
- 4) Table 11, “Valid values for datetime fields”, specifies the constraints on the values of the <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 11—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	-12 to 13
TIMEZONE_MINUTE	-59 to 59

NOTE 62 – 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

6.1 <data type>

“leap seconds”. Interval arithmetic that involves leap seconds or discontinuities in calendars will produce implementation-defined results.

- 5) If WITH TIME ZONE is not specified, then the time zone displacement of the datetime data type is effectively the current default time zone displacement of the SQL-session.
- 6) An interval value can be zero, positive, or negative.
- 7) 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 12, “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 12—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>.

- 8) If <data type> is a <collection type>, then a collection type descriptor is created. The collection type descriptor includes an indication of the <collection type constructor> specified by the <collection type> and the descriptor of the element type of the <collection type>.
- 9) For a <row type> RT , the degree of RT is initially set to zero. The General Rules of Subclause 6.2, “<field definition>”, specify the degree of RT during the definition of the fields of RT .
- 10) 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>.
- 11) A <user-defined type name> identifies a user-defined type.
- 12) A <reference type> identifies a reference type.
- 13) If <data type> is a <reference type>, then a reference type descriptor is created. The reference type descriptor includes the name of the <referenced type>. If a <scope clause> is specified, then the reference type descriptor includes STN , identifying the scope of the reference type.

Conformance Rules

- 1) Without Feature S023, “Basic structured types”, <user-defined type name>, if specified, shall not identify a structured type.
- 2) Without Feature T031, “BOOLEAN data type”, a <predefined type> shall not be a <boolean type>.
- 3) Without Feature F555, “Enhanced seconds precision”, a <time precision>, if specified, shall specify 0 (zero).
- 4) Without Feature F555, “Enhanced seconds precision”, a <timestamp precision>, if specified, shall specify 0 (zero) or 6.
- 5) Without Feature F511, “BIT data type”, a <data type> shall not be a <bit string type>.
- 6) Without Feature F052, “Intervals and datetime arithmetic”, a <data type> shall not be an <interval type>.
- 7) Without Feature F421, “National character”, a <data type> shall not be a <national character string type>.
- 8) Without Feature F451, “Character set definition”, or Feature F461, “Named character sets”, a <data type> shall not specify CHARACTER SET.
- 9) Without Feature F421, “National character”, a <national character string type> shall not specify NATIONAL CHARACTER LARGE OBJECT, NCHAR LARGE OBJECT, or NCLOB.
- 10) Without Feature F411, “Time zone specification”, a <datetime data type> shall not specify <with or without time zone>.
- 11) Without Feature S041, “Basic reference types”, conforming SQL language shall not specify <reference type>.
- 12) Without Feature T051, “Row types”, conforming SQL language shall not specify <row type>.
- 13) Without Feature S091, “Basic array support”, conforming SQL language shall not specify <collection type>.
- 14) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not specify a <scope clause> that is not simply contained in a <data type> that is simply contained in a <column definition>.
- 15) Without Feature S092, “Arrays of user-defined types”, the <data type> simply contained in a <collection type> shall not be a <user-defined type>.
- 16) Without Feature S094, “Arrays of reference types”, the <data type> simply contained in a <collection type> shall not be a <reference type>.
- 17) Without Feature T041, “Basic LOB data type support”, conforming SQL language shall not specify LARGE OBJECT, BLOB, CLOB, or NCLOB.

6.2 <field definition>

Function

Define a field of a row type.

Format

```
<field definition> ::=  
    <field name>  
    <data type>  
    [ <reference scope check> ]  
    [ <collate clause> ]
```

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) If the declared type of the field is character string, then the collation of the field is
Case:
 - a) If <collate clause> is specified, then the collation specified by that <collate clause>.
 - b) Otherwise, the default collation of the character set of the field.
- 5) If <data type> is a <reference type> that contains a <scope clause>, then a <reference scope check> that specifies either REFERENCES ARE NOT CHECKED or REFERENCES ARE CHECKED ON DELETE NO ACTION shall be specified; otherwise, <reference scope check> shall not be specified.
- 6) Let *DT* be the <data type>.
- 7) 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>.
- 8) If *DT* is a <character string type> that identifies a character set that specifies a <collate clause> and the <field definition> does not contain a <collate clause>, then the <collate clause> of the <character string type> is implicit in the <field definition>.
- 9) If <collate clause> is specified, then the declared type shall be a character string type.

Access Rules

- 1) If a <data type> is specified that is a user-defined type *U*, then the applicable privileges of the <authorization identifier> of the schema or SQL-client module that contains the <field definition> shall include USAGE on *U*.

NOTE 63 – “applicable privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) The <collate clause> specifies the default collating sequence for the field. If <collate clause> is not specified, then the default collating sequence is that used for comparisons of Coercible coercibility characteristic, as defined in Subclause 8.2, "<comparison predicate>".
- 2) A data type descriptor is created that describes the declared type of the field being defined.
- 3) The degree of the row type *RT* being defined in the simply containing <row type> is increased by 1 (one).
- 4) 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) If the <field definition> contains a <collate clause>, then the <collation name> of the <collate clause>.
 - d) The field descriptor is included in the row type descriptor for *RT*.
 - e) If <data type> is a reference type, then whether references are checked.

Conformance Rules

- 1) Without Feature T051, "Row types", conforming SQL language shall not contain any <field definition>.
- 2) Without Feature F691, "Collation and translation", and Feature T051, "Row types", a <field definition> shall not contain a <collate clause>.
- 3) Without Feature S043, "Enhanced reference types", conforming SQL language shall not specify REFERENCES ARE CHECKED.

6.3 <value specification> and <target specification>

6.3 <value specification> and <target specification>**Function**

Specify one or more values, host parameters, or SQL parameters.

Format

```
<value specification> ::=
    <literal>
  | <general value specification>
```

```
<unsigned value specification> ::=
    <unsigned literal>
  | <general value specification>
```

```
<general value specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | CURRENT_PATH
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
  | SYSTEM_USER
  | USER
  | VALUE
```

```
<simple value specification> ::=
    <literal>
  | <host parameter name>
  | <SQL parameter reference>
```

```
<target specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <column reference>
```

```
<simple target specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <column reference>
```

```
<host parameter specification> ::=
    <host parameter name> [ <indicator parameter> ]
```

```
<indicator parameter> ::=
    [ INDICATOR ] <host parameter name>
```

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>.

6.3 <value specification> and <target specification>

- 3) If USER is specified, then CURRENT_USER is implicit.

NOTE 64 – In an environment where the SQL-implementation conforms to Core SQL, conforming SQL language that contains either:

A specified or implied <comparison predicate> that compares the <value specification> USER with a <value specification> other than USER, or

A specified or implied assignment in which the “value” (as defined in Subclause 9.2, “Store assignment”) contains the <value specification> USER

will become non-conforming in an environment where the SQL-implementation conforms to some SQL package that supports character internationalization, unless the character repertoire of the implementation-defined character set in that environment is identical to the character repertoire of SQL_IDENTIFIER.

- 4) The declared type of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, and CURRENT_PATH is character string. Whether the character string is fixed length or variable length, and its length if it is fixed length or maximum length if it is variable length, are implementation-defined. The character set of the character string is SQL_IDENTIFIER.
- 5) 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.
- 6) If the declared type of the <value specification> or <unsigned value specification> is character string, then the <value specification> or <unsigned value specification> has the *Coercible* coercibility characteristic, and the collating sequence is determined by Subclause 4.2.3, “Rules determining collating sequence usage”.
- 7) A <target specification> or <simple target specification> that is a <column reference> shall be a new transition variable column reference.

NOTE 65 – “New transition variable column reference” is defined in Subclause 6.5, “<identifier chain>”. 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 expression> that simply contains a <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 <value expression primary>, then

Case:

- i) If *BVE* is of the form “<left paren> <value expression> <right paren>” and the <value expression> is a known-not-null condition for *X*, then *BVE* is a *known-not-null condition* for *X*.

- ii) Otherwise, *BVE* is not a known-not-null condition for *X*.

6.3 <value specification> and <target specification>

- c) 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*.
- d) 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 66 – For simplicity, the rules do not attempt to analyze conditions such as “NOT NOT A IS NULL”, or “NOT (A IS NULL OR NOT (B = 2))”
- e) 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 a *known-not-null condition* for *X*.
ii) Otherwise, *BVE* is not a known-not-null condition for *X*.
f) If *BVE* is of the form “*BVE1* OR *BVE2*”, then *BVE* is not a known-not-null condition for *X*.
NOTE 67 – 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.

Access Rules

None.

General Rules

- 1) A <value specification> or <unsigned value specification> specifies a value that is not selected from a table.
- 2) A <host parameter specification> identifies a host parameter or a host parameter and an indicator parameter in an <SQL-client module definition>.
- 3) A <target specification> specifies a host parameter, an output SQL parameter, or the column of a new transition variable.
- 4) If a <host parameter specification> contains an <indicator parameter> and the value of the indicator parameter is negative, then the value specified by the <host parameter specification> is null; otherwise, the value specified by a <host parameter specification> is the value of the host parameter identified by the <host parameter name>.
- 5) The value specified by a <literal> is the value represented by that <literal>.
- 6) The value specified by CURRENT_USER is the value of the current user identifier.
- 7) The value specified by SESSION_USER is the value of the SQL-session user identifier.

6.3 <value specification> and <target specification>

- 8) The value specified by CURRENT_ROLE is the value of the current role name.
- 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 SQL-statement whose execution caused the SYSTEM_USER <general value specification> to be evaluated.
- 10) The value specified by CURRENT_PATH is a <schema name list> where <catalog name>s are <delimited identifier>s and the <unqualified schema name>s are <delimited identifier>s. Each <schema name> is separated from the preceding <schema name> by a <comma> with no intervening <space>s. The schemas referenced in this <schema name list> are those referenced in the SQL-path of the current SQL-session context, in the order in which they appear in that SQL-path.
- 11) If a <simple value specification> evaluates to the null value, then an exception condition is raised: *data exception — null value not allowed*.
- 12) A <simple target specification> specifies a host parameter, an output SQL parameter, or a column of a new transition variable. A <simple target specification> can only be assigned a value that is not null.
- 13) 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*.

Conformance Rules

- 1) Without Feature S071, “SQL paths in function and type name resolution”, a <general value specification> shall not specify CURRENT_PATH.
- 2) Without Feature F251, “Domain support”, a <general value specification> shall not specify VALUE.
- 3) Without Feature F321, “User authorization”, a <general value specification> shall not specify CURRENT_USER, SYSTEM_USER, or SESSION_USER.
NOTE 68 – Although CURRENT_USER and USER are semantically the same, in Core SQL, CURRENT_USER must be specified as USER.
- 4) Without Feature T332, “Extended roles”, conforming SQL language shall not specify CURRENT_ROLE.

6.4 <contextually typed value specification>**6.4 <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>
```

```
<default specification> ::=
    DEFAULT
```

Syntax Rules

- 1) The declared type *DT* of an <empty specification> *ES* is *ET* ARRAY[0], where the element type *ET* is determined by the context in which *ES* appears. *ES* is effectively replaced by CAST (*ES* AS *DT*).

NOTE 69 – 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 70 – 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 71 – 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.

6.4 <contextually typed value specification>**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”, <empty specification> shall not be specified.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

6.5 <identifier chain>

Function

Disambiguate a <period>-separated chain of identifiers.

Format

```
<identifier chain> ::=  
    <identifier> [ { <period> <identifier> }... ]
```

```
<basic identifier chain> ::=  
    <identifier chain>
```

Syntax Rules

- 1) Let IC be an <identifier chain>.
- 2) Let N be the number of <identifier>s immediately contained in IC .
- 3) Let I_i , $1 \text{ (one)} \leq i \leq N$, 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 $PIC_j = PIC_{j-1} \text{ <period> } I_j$. 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 of a table is said to be *refinable* if the data type of C is a row type or a structured type.
- 7) 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 $N = 1 \text{ (one)}$, then IC shall be contained within the scope of one or more exposed <table or query name>s or <correlation name>s whose associated tables include a column whose <identifier> is equivalent to I_1 or within the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter whose <identifier> is equivalent to I_1 . Let the phrase *possible scope tags* denote those exposed <table name>s, <correlation name>s, and <routine name>s.
Case:
 - i) If the number of possible scope tags in the innermost scope containing a possible scope tag is 1 (one), then
Case:
 - 1) If the innermost possible scope tag is a <table or query name> or <correlation name>, then let T be the table associated with the possible scope tag, and let C be the column of T whose <identifier> 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 T , and the basis referent is C .

- 2) If the innermost possible scope tag is a <routine name>, then let SP be the SQL parameter whose <identifier> 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 .
- ii) Otherwise, each possible scope tag shall be a <table or query name> or a <correlation name> of a <table reference> 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 the basis of IC , the basis length is 1 (one), and the basis referent is C .
NOTE 72 – “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 within the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter SP whose <identifier> 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 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 whose <identifier> is equivalent to I_2 , then 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 .
- iii) 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 refinable column C whose <identifier> is equivalent to I_2 , then 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 .
- iv) 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. If the table T associated with EN has a column C whose <identifier> is equivalent to I_N , then 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 .
- v) 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 .
- 8) Let BL be the basis length of IC .
- 9) If $BL < N$, then let TIC be the <value expression primary>:

$$(PIC_{BL}) <period> I_{BL+1} <period> \dots <period> I_N$$

The Syntax Rules of Subclause 6.23, “<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 73 – 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 $I_{BL+1}, \dots, I_N$ are parsed using the Syntax Rules of <field reference> and <method invocation>.

6.5 <identifier chain>

- 10) A <basic identifier chain> shall be an <identifier chain> whose basis is the entire identifier chain.
- 11) A <basic identifier chain> whose basis referent 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 values correlation name> of the <trigger definition>, then the column reference is a *new transition variable column reference*.
- 12) A <basic identifier chain> whose basis referent is an SQL parameter is an *SQL parameter reference*.
- 13) The data type of a <basic identifier chain> *BIC* is the data type of the basis referent of *BIC*.
- 14) If the data type of a <basic identifier chain> *BIC* is character string, then *BIC* has the *Implicit* coercibility attribute, and its collating sequence is the default collating sequence of the basis referent of *BIC*.

Access Rules

None.

General Rules

- 1) Let *BIC* be a <basic identifier chain>.
- 2) If *BIC* is a general column reference, then *BIC* references the column *C* that is the basis referent of *BIC* in a given row of the table that contains *C*.
- 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

None.

6.6 <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 *M*, and shall identify a declared local temporary table *DLTT* of *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* is some <SQL procedure statement>, <trigger definition>, or <table reference> that contains *TE*, then *CR* is an *outer reference* to the qualifying table of *CR*.
- 5) The data type of a <column reference> is the data type of the column that it references.
- 6) If the data type of a <column reference> is character string, then it has the *Implicit* coercibility attribute, and its collating sequence is the default collating sequence of the column that it references.

Access Rules

- 1) If *CR* is a <column reference> whose qualifying table is a base table or a viewed table and that is contained in any of:
 - A <query expression> simply contained in a <cursor specification>, a <view definition> or an <insert statement>.
 - A <sort specification list> contained in a <cursor specification>.
 - A <table expression> immediately contained in a <select statement: single row>.
 - A <search condition> immediately contained in a <trigger definition>, a <delete statement: searched> or an <update statement: searched>.
 - A <select list> immediately contained in a <select statement: single row>.

— A <value expression> simply contained in a <row value expression> immediately contained in a <set clause>.

then let *C* be the column referenced by *CR*.

Case:

a) If <column reference> is contained in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT for *C*.

b) Otherwise, the current privileges shall include SELECT on *C*.

NOTE 74 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

2) If *CR* is a <column reference> that is contained in a <search condition> immediately contained in an <assertion definition>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include REFERENCES on the column referenced by *CR*.

General Rules

None.

Conformance Rules

1) Without Feature F821, “Local table references”, conforming SQL language shall not specify MODULE.

6.7 <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 data type of an <SQL parameter reference> is the data type of the SQL parameter that it references.
- 3) If the data type of an <SQL parameter reference> is character string, then it has the *Implicit* coercibility attribute, and its collating sequence is the default collating sequence of the SQL parameter that it references.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

6.8 <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 be the name of 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 contain no <field reference>.

6.9 <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:

$$VEP \rightarrow QI$$
 - b) Otherwise, <attribute or method reference> is effectively replaced by a <method reference> *AMR* of the form:

$$VEP \rightarrow QI \text{ } SAL$$
- 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 contain no <attribute or method reference>.

6.10 <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 6.11, "<method invocation>", are applied to the <method invocation>:

DEREF (*VEP*) . *MN MRAL*

yielding subject routine *SR* and static SQL argument list *SAL*.

- 4) The data type of <method reference> is the data type of the expression:

DEREF (*VEP*) . *MN MRAL*

Access Rules

- 1) Let *SCOPE* be the table that is the scope of *VEP*.

Case:

- a) If <method reference> is contained 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 6.11, "<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 contain no <method reference>.

6.11 <method invocation>

Function

Reference an SQL-invoked method of a user-defined type value.

Format

```

<method invocation> ::=
    <direct invocation>
  | <generalized invocation>

<direct invocation> ::=
    <value expression primary> <period> <method name>
    [ <SQL argument list> ]

<generalized invocation> ::=
    <left paren> <value expression primary>
    AS <data type> <right paren> <period> <method name>
    [ <SQL argument list> ]

<method selection> ::= <routine invocation>

```

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) If <method invocation> is not immediately contained in <new invocation>, then *MN* shall not be equivalent to the <qualified identifier> of the <user-defined type name> of *UDT*.
- 4) Case:
 - a) If <SQL argument list> is specified, then let *AL* be:

$$A_1, \dots, A_n$$
 where A_i , $1 \text{ (one)} \leq i \leq n$, are the <SQL argument>s immediately contained in <SQL argument list>, taken in order of their ordinal position in <SQL argument list>.
 - b) Otherwise, let *AL* be a zero-length string.
- 5) 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*.
- 6) If <generalized invocation> is specified, then:
 - a) Let *GE* be a <generalized expression> that immediately contains *VEP* and *DT* as <value expression primary> and <user-defined type name>, respectively. Let *RI* be the following <method selection>:

$$MN (GE, AL)$$

b) Otherwise, let *RI* be the following <method selection>:

MN (VEP, AL)

- 7) The Syntax Rules of Subclause 10.4, "<routine invocation>", are applied with *RI* and *TP* as the <routine invocation> and SQL-path, respectively, yielding subject routine *SR* and static SQL argument list *SAL*.

Access Rules

None.

General Rules

- 1) The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SR* and *SAL* as the subject routine and SQL argument list, respectively, yielding value *V* that is the result of the <routine invocation>.
- 2) The value of <method invocation> is *V*.

Conformance Rules

- 1) Without Feature S023, "Basic structured types", conforming SQL language shall contain no <method invocation>.

6.12 <static method invocation>

Function

Invoke a static method.

Format

<static method invocation> ::=
 <user-defined type> <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 <user-defined type> and let *T* be the user-defined type identified by *TN*.
- 2) Let *MN* be the <method name> immediately contained in <static method invocation>.
- 3) Case:
 - a) If <SQL argument list> is specified, then let *AL* be that <SQL argument list>.
 - b) Otherwise, let *AL* be <left paren> <right paren>.
- 4) Let *TP* be an SQL-path containing only the <schema name> of every schema that includes a descriptor of a supertype of *T*.
- 5) Let *RI* be the following <routine invocation>:

MN AL

- 6) Let *SMS* be the following <static method selection>:

RI

- 7) The Syntax Rules of Subclause 10.4, "<routine invocation>", are applied with *RI* as the <routine invocation> immediately contained in the <static method selection> *SMS*, with *TP* as the SQL-path, and with *T* as the user-defined type of the static SQL-invoked method, yielding subject routine *SR* and static SQL argument list *SAL*.

Access Rules

None.

General Rules

- 1) The General Rules of Subclause 10.4, "<routine invocation>", are applied with *SR* and *SAL* as the subject routine and SQL argument list, respectively, yielding a value *V* that is the result of the <routine invocation>.
- 2) The value of <static method invocation> is *V*.

Conformance Rules

- 1) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain any <static method invocation>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

6.13 <element reference>

Function

Return an element of an array.

Format

```

<element reference> ::=
    <array value expression>
    <left bracket or trigraph> <numeric value expression> <right bracket or trigraph>

```

Syntax Rules

- 1) The declared type of an <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 <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 <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 any <element reference>.

6.14 <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 of *RT*.
- 3) The declared type of the result of the <dereference operation> is the declared type of *AN*.
- 4) Let *S* be the name of the referenceable table in the scope of the reference type of *RVE*.
 - a) Let *OID* be the name of the self-referencing column of *S*.
 - b) <dereference operation> is equivalent to a <scalar subquery> of the form:

```
( SELECT AN  
  FROM S  
  WHERE S.OID = VEP )
```

Access Rules

None.

General Rules

None.

Conformance Rules

- 1) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain any <dereference operation>.

6.15 <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 scope included in the declared type of RVE shall not be empty.
- 2) The declared type of RR is the structured type that is referenced by the declared type of RVE .
- 3) Let m be the number of subtables of the table referenced by the scope table name $SCOPE$ included in the declared type of RVE . Let S_i , $1 \text{ (one)} \leq i \leq m$, be the subtables, arbitrarily ordered, of $SCOPE$.
- 4) For each S_i , $1 \text{ (one)} \leq i \leq m$, let STN_i be the name included in the descriptor of S_i of the structured type ST_i associated with S_i , let $REFCOL_i$ be the self-referencing column of S_i , let N_i be the number of attributes of ST_i , and let A_{ij} , $1 \text{ (one)} \leq j \leq N_i$, be the names of the attributes of ST_i , therefore also the names of the columns of S_i .
- 5) $SCOPE$ is called the *scoped table* of RR .

Access Rules

- 1) Case:
 - a) If <reference resolution> is contained in a <schema definition>, then the applicable privileges of 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 include SELECT WITH HIERARCHY OPTION on at least one supertable of $SCOPE$.

NOTE 75 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) The value of <reference resolution> is the value of:

```
(
  SELECT A1,1 ( . . . A1,N1
    ( STN1() , A1,N1 ), . . . A1,1 )
  FROM ONLY S1
  WHERE S1.REFCOL1 = RVE
  UNION
  SELECT A2,1 ( . . . A2,N2
    ( STN2() , A2,N2 ), . . . A2,1 )
  FROM ONLY S2
  WHERE S2.REFCOL2 = RVE
  UNION
  .
  .
  .
  UNION
  SELECT Am,1 ( . . . Am,Nm
    ( STNm() , Am,Nm ), . . . Am,1 )
  FROM ONLY Sm
  WHERE Sm.REFCOLm = RVE
)
```

NOTE 76 – 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 any <reference resolution>.

6.16 <set function specification>

Function

Specify a value derived by the application of a function to an argument.

Format

```

<set function specification> ::=
    COUNT <left paren> <asterisk> <right paren>
    | <general set function>
    | <grouping operation>

<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

<grouping operation> ::=
    GROUPING <left paren> <column reference> <right paren>

<set quantifier> ::=
    DISTINCT
    | ALL

```

Syntax Rules

- 1) If <set quantifier> is not specified, then ALL is implicit.
- 2) The argument of COUNT(*) and the argument source of a <general set function> is a table or a group of a grouped table as specified in Subclause 7.10, "<having clause>", and Subclause 7.11, "<query specification>".
- 3) Let T be the argument or argument source of a <set function specification>.
- 4) The <value expression> simply contained in <set function specification> shall not contain a <set function specification> or a <subquery>. If the <value expression> contains a column reference that is an outer reference, then that outer reference shall be the only column reference contained in the <value expression>.
NOTE 77 – *Outer reference* is defined in Subclause 6.6, "<column reference>".
- 5) If a <set function specification> contains a column reference that is an outer reference, then the <set function specification> shall be contained in either:
 - a) A <select list>.

6.16 <set function specification>

- b) A <subquery> of a <having clause>, in which case the qualifying table of the <column reference> shall be the table referenced by a <table reference> that is directly contained in the <table expression> that directly contains the <having clause>.

NOTE 78 – *Outer reference* and “qualifying scope” are defined in Subclause 6.6, “<column reference>”.

- 6) Let *DT* be the declared type of the <value expression>.
- 7) If the <set function specification> specifies a <general set function> whose <set qualifier> is DISTINCT and *DT* is a user-defined type, then the comparison form of *DT* shall be FULL.
- 8) If the <set function specification> specifies a <set function type> that is MAX or MIN and *DT* is a user-defined type, then the comparison form of *DT* shall be FULL.
- 9) If the <set function specification> specifies a <set function type> that is AVG or SUM, then *DT* shall not be a <collection type>, row type, user-defined type, reference type, or large object string type.
- 10) If the <set function specification> specifies a <set function type> that is MAX or MIN, then *DT* shall not be a <collection type>, row type, reference type, or large object string type.
- 11) If EVERY, ANY, or SOME is specified, then *DT* shall be boolean and the declared type of the result is boolean.
- 12) If COUNT is specified, then the declared type of the result is exact numeric with implementation-defined precision and scale of 0 (zero).
- 13) If MAX or MIN is specified, then the declared type of the result is *DT*.
- 14) If SUM or AVG is specified, then:
 - a) *DT* shall be a numeric type or an interval type.
 - b) If SUM is specified and *DT* is exact numeric with scale *S*, then the declared type of the result is exact numeric with implementation-defined precision and scale *S*.
 - c) If AVG is specified and *DT* is exact numeric, then the declared type of the result is exact numeric with implementation-defined precision not less than the precision of *DT* and implementation-defined scale not less than the scale of *DT*.
 - d) If *DT* is approximate numeric, then the declared type of the result is approximate numeric with implementation-defined precision not less than the precision of *DT*.
 - e) If *DT* is interval, then the declared type of the result is interval with the same precision as *DT*.
- 15) If a <grouping operation> is specified, then:
 - a) <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).
- 16) If the declared type of the result is character string, then the collating sequence and the coercibility characteristic are determined as in Subclause 4.2.3, “Rules determining collating sequence usage”.

Access Rules

None.

General Rules

1) Case:

- a) If COUNT(*) is specified, then the result is the cardinality of *T*.
- b) Otherwise, let *TX* be the single-column table that is the result of applying the <value expression> to each row of *T* 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*.

2) Case:

- a) 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.
- b) Otherwise, let *TXA* be *TX*.

3) Case:

- a) If the <general set function> COUNT is specified, then the result is the cardinality of *TXA*.
- b) If *TXA* is empty, then the result is the null value.
- c) If AVG is specified, then the result is the average of the values in *TXA*.
- d) 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.
- e) 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*.

f) If EVERY is specified, then

Case:

- i) If the value of some element of *TXA* is false, then the result is false.
- ii) Otherwise, the result is true.

g) If ANY or SOME is specified, then

Case:

- i) If the value of some element of *TXA* is true, then the result is true.
- ii) Otherwise, the result is false.

- h) If <grouping operation> is specified, then:
 - i) Let *CR* be the <column reference> contained in *GO*.
 - ii) Let *COL1* be the column that is referenced by *CR*.
 - iii) Let *COL2* be the column corresponding to *COL1* in the group-by result *GBR* of the innermost <query specification> that contains *GO*.
 - iv) Let *GCN* be the ordinal position of *COL2* in the group-by result.
 - v) Let *COL3* be the column in *GBR* whose ordinal position is *GCN*+1 (one).
 - vi) The result of the <grouping operation> is the value of *COL3*.
- i) If <grouping operation> is specified, then let *GCN* be the ordinality of the column in the group-by result that is referenced by the <column reference> contained in the <grouping operation>. The result of the <grouping operation> is the value of that column in the group-by result whose ordinality is *GCN*+1.

Conformance Rules

- 1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <set function type> that specifies EVERY, ANY, or SOME.
- 2) Without Feature F561, "Full value expressions", or Feature F801, "Full set function", if a <general set function> specifies DISTINCT, then the <value expression> shall be a column reference.
- 3) Without Feature F441, "Extended set function support", if a <general set function> specifies or implies ALL, then COUNT shall not be specified.
- 4) Without Feature F441, "Extended set function support", if a <general set function> specifies or implies ALL, then the <value expression> shall contain a column reference that references a column of *T*.
- 5) Without Feature F441, "Extended set function support", if the <value expression> contains a column reference that is an outer reference, then the <value expression> shall be a column reference.
- 6) Without Feature F441, "Extended set function support", no column reference contained in a <set function specification> shall reference a column derived from a <value expression> that generally contains a <set function specification> *SFS2* without an intervening <routine invocation>.
- 7) Without Feature T431, "CUBE and ROLLUP", conforming SQL language shall not contain a <set function specification> that is a <grouping operation>.
- 8) Without Feature S024, "Enhanced structured types", in a <general set function>, if MAX or MIN is specified, then the <value expression> shall not be of a structured type.
- 9) Without Feature S024, "Enhanced structured types", the declared type of a <general set function> shall not be structured type.

6.17 <numeric value function>

Function

Specify a function yielding a value of type numeric.

Format

```

<numeric value function> ::=
    <position expression>
    | <extract expression>
    | <length expression>
    | <cardinality expression>
    | <absolute value expression>
    | <modulus expression>

<position expression> ::=
    <string position expression>
    | <blob position expression>

<string position expression> ::=
    POSITION <left paren> <string value expression>
    IN <string value expression> <right paren>

<blob position expression> ::=
    POSITION <left paren> <blob value expression>
    IN <blob value expression> <right paren>

<length expression> ::=
    <char length expression>
    | <octet length expression>
    | <bit length expression>

<char length expression> ::=
    { CHAR_LENGTH CHARACTER_LENGTH }
    <left paren> <string value expression> <right paren>

<octet length expression> ::=
    OCTET_LENGTH <left paren> <string value expression> <right paren>

<bit length expression> ::=
    BIT_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>

Syntax Rules

- 1) If <position expression> is specified, then the declared type of the result is exact numeric with implementation-defined precision and scale 0 (zero).
- 2) If <string position expression> is specified, then both <string value expression>s shall be <bit value expression>s or both shall be <character value expression>s having the same character repertoire.
- 3) If <extract expression> is specified, then
 Case:
 - a) If <extract field> is a <primary datetime field>, then it shall identify a <primary date-time 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.
- 4) If <extract expression> is specified, then
 Case:
 - a) If <primary datetime field> does not specify SECOND, then the declared type of the result is exact numeric with implementation-defined precision and scale 0 (zero).
 - b) Otherwise, the declared type of the result is exact numeric with implementation-defined precision and scale. The implementation-defined scale shall not be 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>.
- 5) If a <length expression> is specified, then the declared type of the result is exact numeric with implementation-defined precision and scale 0 (zero).
- 6) If <cardinality expression> is specified, then the declared type of the result is exact numeric with implementation-defined precision and scale 0 (zero).
- 7) If <absolute value expression> is specified, then the declared type of the result is the declared type of the immediately contained <numeric value expression>.

- 8) 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>.

Access Rules

None.

General Rules

- 1) If the value of one or more <string value expression>s, <datetime value expression>s, <interval value expression>s, and <collection value expression>s that are simply contained in a <numeric value function> is the null value, then the result of the <numeric value function> is the null value.
- 2) If <string position expression> is specified, then
Case:
 - a) If the first <string value expression> has a length of 0 (zero), then the result is 1 (one).
 - b) If the value of the first <string value expression> is equal to an identical-length substring of contiguous characters or bits from the value of the second <string value expression>, then the result is 1 (one) greater than the number of characters or bits within the value of the second <string value expression> preceding the start of the first such substring.
 - c) Otherwise, the result is 0 (zero).
- 3) If <blob position expression> is specified, then:
Case:
 - a) If the first <blob value expression> has a length of 0 (zero), then the result is 1 (one).
 - b) If the value of the first <blob value expression> is equal to an identical-length substring of contiguous octets from the value of the second <blob value expression>, then the result is 1 (one) greater than the number of octets within the value of the second <blob value expression> preceding the start of the first such substring.
 - c) Otherwise, the result is 0 (zero).
- 4) If <extract expression> is specified, then
Case:
 - a) If <extract field> is a <primary datetime field>, then the result is the value of the datetime field identified by that <primary datetime field> and has the same sign as the <extract source>.
NOTE 79 – 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 associated with the <datetime value expression>. If <extract field> is TIMEZONE_HOUR, then the result is calculated as

EXTRACT (HOUR FROM *TZ*)

Otherwise, the result is calculated as

EXTRACT (MINUTE FROM TZ)

- 5) If a <char length expression> is specified, then let S be the <string value expression>.

Case:

- a) If the most specific type of S is character string, then the result is the number of characters in the value of S .

NOTE 80 – The number of characters in a character string is determined according to the semantics of the character set of that character string.

- b) Otherwise, the result is OCTET_LENGTH(S).

- 6) If an <octet length expression> is specified, then let S be the <string value expression>. The result of the <octet length expression> is the smallest integer not less than the quotient of the division (BIT_LENGTH(S)/8).

- 7) If a <bit length expression> is specified, then let S be the <string value expression>. The result of the <bit length expression> is the number of bits in the value of S .

- 8) The result of <cardinality expression> is the number of elements of the result of the <collection value expression>.

- 9) 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*.

- 10) If <modulus expression> is specified, then let N be the value of the immediately contained <numeric value expression dividend> and let M be the value of the immediately contained <numeric value expression divisor>.

Case:

- a) If either N or M is the null value, then the result is the null value.

- b) If M is zero, then an exception condition is raised: *data exception — division by zero*.

- c) Otherwise, the result is the unique nonnegative exact numeric value R with scale 0 (zero) such that all of the following are true:

- i) R has the same sign as N .

- ii) The absolute value of R is less than the absolute value of M .

- iii) $N = M * K + R$ for some exact numeric value K with scale 0 (zero).

Conformance Rules

- 1) Without Feature S091, “Basic array support”, a <numeric value function> shall not be a <cardinality expression>.
- 2) Without Feature F052, “Intervals and datetime arithmetic”, a <numeric value function> shall not be an <extract expression>.
- 3) Without Feature F052, “Intervals and datetime arithmetic”, and Feature F411, “Time zone specification”, a <numeric value function> shall not be an <extract expression> that specifies a <time zone field>.
- 4) Without Feature F421, “National character”, a <string value expression> simply contained in a <length expression> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.
- 5) Without Feature T441, “ABS and MOD functions”, conforming language shall not specify ABS or MOD.

6.18 <string value function>

Function

Specify a function yielding a value of type character string or bit string.

Format

```
<string value function> ::=
    <character value function>
    | <blob value function>
    | <bit value function>

<character value function> ::=
    <character substring function>
    | <regular expression substring function>
    | <fold>
    | <form-of-use conversion>
    | <character translation>
    | <trim function>
    | <character overlay function>
    | <specific type method>

<character substring function> ::=
    SUBSTRING <left paren> <character value expression> FROM <start position>
    [ FOR <string length> ] <right paren>

<regular expression substring function> ::=
    SUBSTRING <left paren> <character value expression> FROM
    <character value expression> FOR <escape character> <right paren>

<fold> ::= { UPPER | LOWER } <left paren> <character value expression> <right paren>

<form-of-use conversion> ::=
    CONVERT <left paren> <character value expression>
    USING <form-of-use conversion name> <right paren>

<character translation> ::=
    TRANSLATE <left paren> <character value expression>
    USING <translation name> <right paren>

<trim function> ::=
    TRIM <left paren> <trim operands> <right paren>

<trim operands> ::=
    [ [ <trim specification> ] [ <trim character> ] FROM ] <trim source>

<trim source> ::= <character value expression>

<trim specification> ::=
    LEADING
    | TRAILING
    | BOTH

<trim character> ::= <character value expression>

<character overlay function> ::=
    OVERLAY <left paren> <character value expression>
    PLACING <character value expression>
```



```

    FROM <start position>
    [ FOR <string length> ] <right paren>

<specific type method> ::=
    <user-defined type value expression> <period> SPECIFICTYPE

<blob value function> ::=
    <blob substring function>
    | <blob trim function>
    | <blob overlay function>

<blob substring function> ::=
    SUBSTRING <left paren> <blob value expression> FROM <start position>
    [ FOR <string length> ] <right paren>

<blob trim function> ::=
    TRIM <left paren> <blob trim operands> <right paren>

<blob trim operands> ::=
    [ [ <trim specification> ] [ <trim octet> ] FROM ] <blob trim source>

<blob trim source> ::= <blob value expression>

<trim octet> ::= <blob value expression>

<blob overlay function> ::=
    OVERLAY <left paren> <blob value expression>
    PLACING <blob value expression>
    FROM <start position>
    [ FOR <string length> ] <right paren>

<bit value function> ::=
    <bit substring function>

<bit substring function> ::=
    SUBSTRING <left paren> <bit 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>, <blob value function>, or <bit value function>. If <string value function> is <character value function>, then the coercibility and collating sequence of <string value function> are the coercibility and collating sequence, respectively, of the simply contained <character value function>.
- 2) The declared type, coercibility, and collating sequence of <character value function> are the declared type, coercibility, and collating sequence, respectively, of the immediately contained <character substring function>, <regular expression substring function>, <fold>, <form-of-use conversion>, <character translation>, <trim function>, or <character overlay function>.
- 3) The declared type of a <start position> and <string length> shall be exact numeric with scale 0 (zero).

- 4) If <character substring function> is specified, then:
 - a) The declared type of the <character substring function> is variable-length character string with maximum length equal to the fixed length or maximum variable length of the <character value expression>. The character repertoire and form-of-use of the <character substring function> are the same as the character repertoire and form-of-use of the <character value expression>.
 - b) The collating sequence and the coercibility characteristic are determined as specified for monadic operators in Subclause 4.2.3, "Rules determining collating sequence usage", where the first operand of <character substring function> plays the role of the monadic operand.
- 5) 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) The declared type of the <regular expression substring function> is variable-length character string with the same character repertoire and with maximum variable length of the first <character value expression>.
 - c) The value of the <escape character> shall have length 1 (one).
 - d) The collating sequence and the coercibility characteristic are determined as specified for monadic operators in Subclause 4.2.3, "Rules determining collating sequence usage", where the first operand of <regular expression substring function> plays the role of the monadic operand.
- 6) If <fold> is specified, then:
 - a) The declared type of the result of <fold> is the declared type of the <character value expression>.
 - b) The collating sequence and the coercibility characteristic are determined as specified for monadic operators in Subclause 4.2.3, "Rules determining collating sequence usage", where the operand of the <fold> is the monadic operand.
- 7) If <form-of-use conversion> is specified, then:
 - a) <form-of-use conversion> 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 <simply value specification> that is a <host parameter name> or a <value specification> that is a <host parameter specification>.
 - b) A <form-of-use conversion name> shall identify a form-of-use conversion.
 - c) The declared type of the result is variable-length character string with implementation-defined maximum length. The character set of the result is the same as the character repertoire of the <character value expression> and form-of-use determined by the form-of-use conversion identified by the <form-of-use conversion name>. Let *CR* be that character repertoire. The result has the *Implicit* coercibility characteristic and its collating sequence is the default collating sequence of *CR*.

8) If <character translation> is specified, then:

- a) A <translation name> shall identify a character translation.
- b) The declared type of the <character translation> is variable-length character string with implementation-defined maximum length and character repertoire equal to the character repertoire of the target character set of the translation. Let *CR* be that character repertoire. The result has the *Implicit* coercibility characteristic and its collating sequence is the default collating sequence of *CR*.

9) If <trim function> is specified, then;

a) If FROM is specified, then:

- i) Either <trim specification> or <trim character> or both shall be specified.
- ii) If <trim specification> is not specified, then BOTH is implicit.
- iii) If <trim character> is not specified, then ' ' is implicit.

b) Otherwise, let *SRC* be <trim source>.

TRIM (*SRC*)

is equivalent to

TRIM (BOTH ' ' FROM *SRC*)

- c) 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>.
- d) If a <trim character> is specified, then <trim character> and <trim source> shall be comparable.
- e) The character repertoire and form-of-use of the <trim function> are the same as those of the <trim source>.
- f) The collating sequence and the coercibility characteristic are determined as specified for monadic operators in Subclause 4.2.3, "Rules determining collating sequence usage", where the <trim source> plays the role of the monadic operand.

10) 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*)

6.18 <string value function>

- 11) 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.
- 12) The declared type of <blob value function> is the declared type of the immediately contained <blob substring function>, <blob trim function>, or <blob overlay function>.
- 13) If <blob substring function> is specified, then the declared type of the <blob substring function> is binary string with maximum length equal to the maximum length of the <blob value expression>.
- 14) If <blob trim function> is specified, then:
 - a) If FROM is specified, then:
 - i) Either <trim specification> or <trim octet> or both shall be specified.
 - ii) If <trim specification> is not specified, then BOTH is implicit.
 - iii) If <trim octet> is not specified, then X'00' is implicit.
 - b) Otherwise, let *SRC* be <trim source>.

`TRIM ( SRC )`
 is equivalent to
`TRIM ( BOTH X'00' FROM SRC )`
 - c) The declared type of the <blob trim function> is binary string with maximum length equal to the maximum length of the <blob trim source>.
- 15) If <blob overlay function> is specified, then:
 - a) Let *BV* be the first <blob value expression>, let *SP* be the <start position>, and let *RS* be the second <blob value expression>.
 - b) If <string length> is specified, then let *SL* be <string length>; otherwise, let *SL* be OCTET_LENGTH(*RS*).
 - c) The <blob overlay function> is equivalent to:

`SUBSTRING ( BV FROM 1 FOR SP - 1 )`
`|| RS`
`|| SUBSTRING ( BV FROM SP + SL )`
- 16) The declared type of <bit value function> is the declared type of the immediately contained <bit substring function>.
- 17) If <bit substring function> is specified, then the declared type of the <bit substring function> is variable-length bit string with maximum length equal to the fixed length or maximum variable length of the <bit value expression>.

Access Rules

1) Case:

- a) If <string value function> is contained in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE for every translation identified by a <translation name> contained in the <string value expression>.
- b) Otherwise, the current privileges shall include USAGE for every translation identified by a <translation name> contained in the <string value expression>.

NOTE 81 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) The result of <string value function> is the result of the immediately contained <character value function>, <blob value function>, or <bit value function>.
- 2) The result of <character value function> is the result of the immediately contained <character substring function>, <regular expression substring function>, <form-of-use conversion>, <character translation>, <trim function>, <character overlay function>, or <specific type method>.
- 3) If <character substring function> is specified, then:
 - a) Let C be the value of the <character value expression>, let LC be the length of C , 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 $LC + 1$ and S .
 - c) If either C , S , or L is the null value, then the result of the <character 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 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 <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) Otherwise, *R* shall contain exactly two occurrences of *E* followed by the <double quote> character. There shall exist three substrings, *R1*, *R2*, and *R3* of *R*, such that *R1*, *R2*, and *R3* are regular expressions and

$'R' = 'R1' \parallel 'E' \parallel '' \parallel 'R2' \parallel 'E' \parallel '' \parallel 'R3'$

is true.

- d) If the predicate

$'C' \text{ SIMILAR TO } 'R1' \parallel 'R2' \parallel 'R3' \text{ ESCAPE } 'E'$

is not true, then the result of the <regular expression substring function> is the null value.

- e) Otherwise, the result *S* of the <regular expression substring function> satisfies the following conditions:
 - i) *S* is a substring of *C* such that there are substrings *S1* and *S2* and the predicate

$'C' = 'S1' \parallel 'S' \parallel 'S2'$

is true.

- ii) The predicate

$'S1' \text{ SIMILAR TO } 'R1' \text{ ESCAPE } 'E'$

is true and, for any proper substring *S* of *S1*, the predicate

$'S' \text{ SIMILAR TO } 'R1' \text{ ESCAPE } 'E'$

is false.

- iii) The predicate

$'S3' \text{ SIMILAR TO } 'R2' \text{ ESCAPE } 'E'$

is true and for any proper substring *S3* of *S*, the predicate

$'S3' \text{ SIMILAR TO } 'R2' \text{ ESCAPE } 'E'$

is false.

- 5) 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) Case:
- i) If UPPER is specified, then the result of the <fold> is 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 the result of the <fold> is a copy of *S* in which every 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.
- 6) If a <character translation> is specified, then
- Case:
- a) If the value of <character value expression> is the null value, then the result of the <character translation> is the null value.
 - b) If <translation name> identifies a translation descriptor whose indication of how the translation is performed specifies an SQL-invoked routine *TR*, then the result of the <character translation> is the result of the invocation of *TR* with a single SQL argument that is the <character value expression> contained in the <character translation>.
 - c) Otherwise, the value of the <character translation> is the value returned by the translation identified by the <existing translation name> specified in the translation descriptor of the translation identified by <translation name>.
- 7) If a <form-of-use conversion> is specified, then
- Case:
- a) If the value of <character value expression> is the null value, then the result of the <form-of-use conversion> is the null value.
 - b) Otherwise, the value of the <form-of-use conversion> is the value of the <character value expression> after the application of the form-of-use conversion specified by <form-of-use conversion name>.
- 8) 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.
- 9) 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 *RV* be:
"*CN*". "*SN*". "*UN*"
 - c) The result of <specific type method> is *RV*.
- 10) The result of <blob value function> is the result of the simply contained <blob substring function>, <blob trim function>, or <blob overlay function>.
- 11) If <blob substring function> is specified, then
- a) Let *B* be the value of the <blob value expression>, let *LB* be the length in octets of *B*, and let *S* be the value of the <start position>.
 - b) If <string length> is specified, then let *L* be the value of <string length> and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LB*+1 and *S*.
 - c) If either *B*, *S*, or *L* is the null value, then the result of the <blob substring function> is the null value.
 - d) If *E* is less than *S*, then an exception condition is raised: *data exception — substring error*.
 - e) Case:
 - i) If *S* is greater than *LB* or if *E* is less than 1 (one), then the result of the <blob substring function> is a zero-length string.
 - ii) Otherwise:
 - 1) Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LB*+1. Let *L1* be *E1*−*S1*.
 - 2) The result of the <blob substring function> is a binary large object string containing *L1* octets of *B* starting at octet number *S1* in the same order that the octets appear in *B*.

- 12) If <blob trim function> is specified, then
- Let S be the value of the <trim source>.
 - Let SO be the value of <trim octet>.
 - If either S or SO the null value, then the result of the <blob trim function> is the null value.
 - If the length in octets of SO is not 1 (one), then an exception condition is raised: *data exception — trim error*.
 - Case:
 - If BOTH is specified or if no <trim specification> is specified, then the result of the <blob trim function> is the value of S with any leading or trailing octets equal to SO removed.
 - If TRAILING is specified, then the result of the <blob trim function> is the value of S with any trailing octets equal to SO removed.
 - If LEADING is specified, then the result of the <blob trim function> is the value of S with any leading octets equal to SO removed.
- 13) The result of <bit value function> is the result of the simply contained <bit substring function>.
- 14) If <bit substring function> is specified, then:
- Let B be the value of the <bit value expression>, let LB be the length in bits of B , and let S be the value of the <start position>.
 - 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 .
 - If either B , S , or L is the null value, then the result of the <bit substring function> is the null value.
 - If E is less than S , then an exception condition is raised: *data exception — substring error*.
 - Case:
 - If S is greater than LB or if E is less than 1 (one), then the result of the <bit substring function> is a zero-length string.
 - Otherwise,
 - Let $S1$ be the larger of S and 1 (one). Let $E1$ be the smaller of E and $LB+1$. Let $L1$ be $E1-S1$.
 - The result of the <bit substring function> is a bit string containing $L1$ bits of B starting at bit number $S1$ in the same order that the bits appear in B .
- 15) 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”, a <string value function> shall not be a <regular expression substring function>.
- 2) Without Feature T312, “OVERLAY function”, conforming SQL language shall not specify a <character overlay function> or a <blob overlay function>.
- 3) Without Feature T042, “Extended LOB data type support”, conforming Core SQL language shall not contain any <blob value function>.
- 4) Without Feature T042, “Extended LOB data type support”, the declared type of a <character value function> shall not be CHARACTER LARGE OBJECT or NATIONAL CHARACTER LARGE OBJECT.
- 5) Without Feature F691, “Collation and translation”, conforming SQL language shall contain no <character translation>.
- 6) Without Feature F691, “Collation and translation”, conforming SQL language shall contain no <form-of-use conversion>.
- 7) Without Feature F511, “BIT data type”, conforming SQL language shall contain no <bit value function>.
- 8) Without Feature F421, “National character”, the <character value expression> simply contained in <character substring function> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.
- 9) Without Feature F421, “National character”, <trim source> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.
- 10) Without Feature S261, “Specific type method”, conforming SQL language shall not specify <specific type method>.

6.19 <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 82 – 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 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 generally contained, without an intervening <routine invocation> whose subject routines do not include an SQL function, in <value expression>s that are contained 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 83 – Activation of triggers is defined in Subclause 4.35.2, “Execution of triggers”.

Conformance Rules

- 1) Without Feature F555, “Enhanced seconds precision”, if LOCALTIME is specified, then <time precision>, if specified, shall be 0 (zero).
- 2) Without Feature F555, “Enhanced seconds precision”, if LOCALTIMESTAMP is specified, then <timestamp precision>, if specified, shall be either 0 (zero) or 6.
- 3) Without Feature F411, “Time zone specification”, CURRENT_TIME and CURRENT_TIMESTAMP shall not be specified.

6.20 <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 expression> 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 expression> 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 Core SQL shall contain no <interval value function>.

6.21 <case expression>

Function

Specify a conditional value.

Format

```
<case expression> ::=
    <case abbreviation>
    | <case specification>

<case abbreviation> ::=
    NULLIF <left paren> <value expression> <comma>
        <value expression> <right paren>
    | COALESCE <left paren> <value expression>
        { <comma> <value expression> }... <right paren>

<case specification> ::=
    <simple case>
    | <searched case>

<simple case> ::=
    CASE <case operand>
        <simple when clause>...
        [ <else clause> ]
    END

<searched case> ::=
    CASE
        <searched when clause>...
        [ <else clause> ]
    END

<simple when clause> ::= WHEN <when operand> THEN <result>

<searched when clause> ::= WHEN <search condition> THEN <result>

<else clause> ::= ELSE <result>

<case operand> ::= <value expression>

<when operand> ::= <value expression>

<result> ::=
    <result expression>
    | NULL

<result expression> ::= <value expression>
```

Syntax Rules

- 1) NULLIF (V_1 , V_2) is equivalent to the following <case specification>:

CASE WHEN $V_1=V_2$ THEN NULL ELSE V_1 END

- 2) COALESCE (V_1 , V_2) is equivalent to the following <case specification>:

CASE WHEN V_1 IS NOT NULL THEN V_1 ELSE V_2 END

- 3) "COALESCE (V_1 , V_2 , . . . , V_n)", for $n \geq 3$, is equivalent to the following <case specification>:

CASE WHEN V_1 IS NOT NULL THEN V_1 ELSE COALESCE (V_2 , . . . , V_n) END

- 4) If a <case specification> specifies a <simple case>, then let *CO* be the <case operand>:

- 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.
- The declared type of each <when operand> *WO* shall be comparable with the declared type of the <case operand>.
- The <case specification> is equivalent to a <searched case> in which each <searched when clause> specifies a <search condition> of the form "*CO=WO*".

- 5) At least one <result> in a <case specification> shall specify a <result expression>.

- 6) If an <else clause> is not specified, then ELSE NULL is implicit.

- 7) The declared type of a <case specification> is determined by applying Subclause 9.3, "Data types of results of aggregations" to the declared types of all <result expression>s in the <case specification>.

Access Rules

None.

General Rules

- 1) Case:

- If a <result> specifies NULL, then its value is the null value.
- If a <result> specifies a <value expression>, then its value is the value of that <value expression>.

- 2) Case:

- If the <search condition> of some <searched when clause> in a <case specification> is true, then the value of the <case specification> is the value of the <result> of the first (leftmost) <searched when clause> whose <search condition> is true, cast as the declared type of the <case specification>.

6.21 <case expression>

- b) If no <search condition> in a <case specification> is true, then the value of the <case expression> is the value of the <result> of the explicit or implicit <else clause>, cast as the declared type of the <case specification>.

Conformance Rules

- 1) Without Feature T042, “Extended LOB data type support”, the declared type of a <result> simply contained in a <case expression> shall not be BINARY LARGE OBJECT or CHARACTER LARGE OBJECT.
- 2) Without Feature F421, “National character”, and Feature T042, “Extended LOB data type support”, the declared type of a <result> simply contained in a <case expression> shall not be NATIONAL CHARACTER LARGE OBJECT.

6.22 <cast specification>

Function

Specify a data conversion.

Format

```

<cast specification> ::=
    CAST <left paren> <cast operand> AS <cast target> <right paren>

<cast operand> ::=
    <value expression> | <implicitly typed value specification>

<cast target> ::=
    <domain name>
    | <data type>

```

Syntax Rules

- 1) Case:
 - a) If a <domain name> is specified, then let *TD* be the <data type> of the specified domain.
 - b) If a <data type> is specified, then let *TD* be the specified <data type>.
- 2) The declared type of the result of the <cast specification> is *TD*.
- 3) If the <cast operand> is a <value expression>, then let *SD* be the declared type of the <value expression>.
- 4) Let *C* be some column and let *CO* be the <cast operand> of a <cast specification> *CS*. *C* is a *leaf column* of *CS* if *CO* consists of a single column reference that identifies *C* or of a single <cast specification> *CS1* for 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> and neither *TD* nor *SD* is a collection type, 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:

6.22 <cast specification>

<data type> SD of <value expression>	<data type> of TD																	
	EN	AN	VC	FC	VB	FB	D	T	TS	YM	DT	BO	UDT	CL	BL	RT	CT	RW
EN	Y	Y	Y	Y	N	N	N	N	N	M	M	N	M	Y	N	M	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N	N	N	M	Y	N	M	N	N
C	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	M	Y	N	M	N	N
B	N	N	Y	Y	Y	Y	N	N	N	N	N	N	M	Y	N	M	N	N
D	N	N	Y	Y	N	N	Y	N	Y	N	N	N	M	Y	N	M	N	N
T	N	N	Y	Y	N	N	N	Y	Y	N	N	N	M	Y	N	M	N	N
TS	N	N	Y	Y	N	N	Y	Y	Y	N	N	N	M	Y	N	M	N	N
YM	M	N	Y	Y	N	N	N	N	N	Y	N	N	M	Y	N	M	N	N
DT	M	N	Y	Y	N	N	N	N	N	N	Y	N	M	Y	N	M	N	N
BO	N	N	Y	Y	N	N	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	M	M	N	N
BL	N	N	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	M	M	N	N
CT	N	N	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	N	N	N

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
 B = Bit String (Fixed- or Variable-length)
 FB = Fixed-length Bit String
 VB = Variable-length Bit String
 D = Date
 T = Time
 TS = Timestamp
 YM = Year-Month Interval
 DT = Day-Time Interval
 BO = Boolean
 UDT = User-Defined Type
 BL = Binary Large Object
 RT = Reference type
 CT = Collection type
 RW = Row type

- 7) If *TD* is an interval and *SD* is exact numeric, then *TD* shall contain only a single <primary datetime field>.
- 8) If *TD* is exact numeric and *SD* is an interval, then *SD* shall contain only a single <primary datetime field>.
- 9) If *SD* is character string and *TD* is fixed-length, variable-length, or large object character string, then the character repertoires of *SD* and *TD* shall be the same.
- 10) If *TD* is a fixed-length, variable-length or large object character string, then the collating sequence of the result of the <cast specification> is the default collating sequence for the character repertoire of *TD* and the result of the <cast specification> has the *Coercible* coercibility characteristic.

- 11) If either *SD* or *TD* is a user-defined type, then 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*.

NOTE 84 – *Source type* is defined in Subclause 4.8, “User-defined types”.

- 12) If either *SD* or *TD* is a reference type, then:
- a) Let *RTSD* and *RTTD* be the referenced types of *SD* and *TD*, respectively.
 - b) If <data type> is specified and contains a <scope clause>, then let *STD* be that scope. Otherwise, let *STD*, possibly empty, be the scope included in the reference type descriptor of *SD*.
 - c) Either *RSTD* and *RTTD* shall be compatible, or there shall be a data type *P* in the type precedence list of *SD* such that all of the following are satisfied:
 - i) There is a user-defined cast CF_P whose user-defined cast descriptor includes *P* as the source data type and *TD* as the target data type.
 - ii) No other data type *Q* that is included as the source data type in the user-defined cast descriptor of some user-defined cast CF_Q that has *TD* as the target data type precedes *P* in the type precedence list of *SD*.

- 13) If *SD* is a collection type, then:

- a) Let *ESD* be the <element type> of *SD*.
- b) Let *ETD* be the <element type> of *TD*.
- c) The <cast specification>
 - a) `CAST (VALUE AS ETD )`

where *VALUE* is a <value expression> of declared type *ESD*, shall be permitted by the Syntax Rules of this Subclause.

- 14) If <domain name> is specified, then let *D* be the domain identified by the <domain name>. If the <cast specification> is not contained in a <schema definition>, then the schema identified by the explicit or implicit qualifier of the <domain name> shall include the descriptor of *D*. If the <cast specification> is contained in a <schema definition> *S*, then *S* shall contain a <schema element> that creates the descriptor of *D*.

Access Rules

- 1) If <domain name> is specified, then

Case:

- a) If <cast specification> is contained in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on the domain identified by <domain name>.
- b) Otherwise, the current privileges shall include USAGE on the domain identified by <domain name>.

NOTE 85 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

- 2) If either *SD* or *TD* is a user-defined type, then

Case:

- a) If <cast specification> is contained in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include EXECUTE on *CF_p*.
- b) Otherwise, the current privileges shall include EXECUTE on *CF_p*.

NOTE 86 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) If the <cast operand> is a <value expression> *VE*, then let *SV* be its value.
- 2) Case:
 - a) If the <cast operand> specifies NULL, then *TV* is the null value.
 - b) If the <cast operand> specifies an <empty specification>, then *TV* is an empty collection of declared type *TD*.
 - c) If *SV* is the null value, then the result is the null value.
 - d) Otherwise, let *TV* be the result of the <cast specification> as specified in the remaining General Rules of this Subclause.
- 3) If either *SD* or *TD* is a user-defined type, then:
 - a) Let *CP* be the cast function contained in the user-defined cast descriptor of *CF_p*.
 - b) The General Rules of Subclause 10.4, “<routine invocation>”, are applied with a static SQL argument list that has a single SQL-argument that is <value expression> and with subject routine *CP*, yielding value *TR* that is the result of the invocation of *CP*.
 - c) *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 *CP_P*.
 - ii) The General Rules of Subclause 10.4, "<routine invocation>", are applied with a static argument list that has a single SQL-argument that is a <value expression> and with subject routine *CP*, yielding value *TV* that is the result of the invocation of *CP*.
 - iii) The scope in the reference type descriptor of *TV* is *STD*.
- 5) If *SD* is an array type, then:
 - a) Let *n* be the number of elements in *SV*. Let *SVE_i* be the *i*-th element of *SV*.
 - b) For *i* varying from 1 (one) to *n*, the following <cast specification> is applied:

CAST (*SVE_i* AS *ETD*)
yielding value *TVE_i*.

- c) *TV* is the array with elements *TVE_i*, 1 (one) ≤ *i* ≤ *n*.
- 6) 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*.
- 7) 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
- $$\text{CAST (} LT \text{ AS } TD \text{)}$$
- 8) 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 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*.
- If *SV* is less than 0 (zero), then let *Y* be the result of
- $$- \text{ } || \text{ } YP$$
- Otherwise, let *Y* be *YP*.
- Case:
- i) If *Y* contains any <SQL language character> that is not in the 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*.

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) If *SV* is less than 0 (zero), then let *Y* be the result of

'-' || *YP*

Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the 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 fixed-length bit string or variable-length bit string, then let *LSV* be the value of *BIT_LENGTH*(*SV*) and let *B* be the *BIT_LENGTH* of the character with the smallest *BIT_LENGTH* in the form-of-use of *TD*. Let *PAD* be the value of the remainder of the division *LSV*/*B*. Let *NC* be a character whose bits all have the value 0 (zero).

If *PAD* is not 0 (zero), then append (*B* − *PAD*) 0-valued bits to the least significant end of *SV*; a completion condition is raised: *warning — implicit zero-bit padding*.

Let *SVC* be the possibly padded value of *SV* expressed as a character string without regard to valid character encodings and let *LTDS* be a character string of *LTD* characters of value *NC* characters in the form-of-use of *TD*.

TV is the result of

SUBSTRING (*SVC* || *LTDS* FROM 1 FOR *LTD*)

Case:

- i) If the length of *TV* is less than the length of *SVC*, then a completion condition is raised: *warning — string data, right truncation*.
- ii) If the length of *TV* is greater than the length of *SVC*, then a completion condition is raised: *warning — implicit zero-bit padding*.
- e) 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 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*.
 - f) 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*.
- 9) 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 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*.

If *SV* is less than 0 (zero), then let *Y* be the result of

'−' || *YP*

Otherwise, let *Y* be *YP*.

Case:

- i) If Y contains any <SQL language character> that is not in the 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*.
- 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) If SV is less than 0 (zero), then let Y be the result of
 '-' || YP
 Otherwise, let Y be YP .
 - iii) Case:
 - 1) If Y contains any <SQL language character> that is not in the 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 fixed-length bit string or variable-length bit string, then let LSV be the value of $BIT\_LENGTH(SV)$ and let B be the BIT_LENGTH of the character with the smallest BIT_LENGTH in the form-of-use of TD . Let PAD be the value of the remainder of the division LSV/B .
- If PAD is not 0 (zero), then append $(B - PAD)$ 0-valued bits to the least significant end of SV ; a completion condition is raised: *warning — implicit zero-bit padding*.
- Let SVC be the possible padded value of SV expressed as a character string without regard to valid character encodings.

Case:

- i) If CHARACTER_LENGTH (SVC) is not greater than MLTD, then TV is SVC.
- ii) Otherwise, TV is the result of

SUBSTRING (SVC FROM 1 FOR MLTD)

If the length of TV is less than the length of SVC, then a completion condition is raised: *warning — string data, right truncation*.

- e) 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 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*.
- f) 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*.

- 10) If TD and SD are binary string data types, then let MLTD be the maximum length in octets of TD.

Case:

- a) If the length in octets of SV is less than or equal to MLTD, then TV is SV.
- b) If the length in octets of SV is larger than MLTD, then TV is the first MLTD octets of SV and a completion condition is raised: *warning — string data, right truncation*.

- 11) If TD is fixed-length bit string, then let LTD be the length in bits of TD. Let BLSV be the result of BIT_LENGTH(SV).

Case:

- a) If BLSV is equal to LTD, then TV is SV expressed as a bit string with a length in bits of BLSV.
- b) If BLSV is larger than LTD, then TV is the first LTD bits of SV expressed as a bit string with a length in bits of LTD, and a completion condition is raised: *warning — string data, right truncation*.

- c) If *BLSV* is smaller than *LTD*, then *TV* is *SV* expressed as a bit string extended on the right with *LTD*–*BLSV* bits whose values are all 0 (zero) and a completion condition is raised: *warning — implicit zero-bit padding*.

- 12) If *TD* is variable-length bit string, then let *MLTD* be the maximum length in bits of *TD*. Let *BLSV* be the result of `BIT_LENGTH(SV)`.

Case:

- a) If *BLSV* is less than or equal to *MLTD*, then *TV* is *SV* expressed as a bit string with a length in bits of *BLSV*.
- b) If *BLSV* is larger than *MLTD*, then *TV* is the first *MLTD* bits of *SV* expressed as a bit string with a length in bits of *MLTD* and a completion condition is raised: *warning — string data, right truncation*.

- 13) If *TD* is the datetime data type `DATE`, then

Case:

- a) If *SD* is character string, then *SV* is replaced by

`TRIM ( BOTH ' ' FROM VE )`

Case:

- i) If the rules for <literal> or for <unquoted date string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.
- ii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format*.
- iii) Otherwise, an exception condition is raised: *data exception — invalid datetime format*.
- b) If *SD* is a date, then *TV* is *SV*.
- c) If *SD* is the datetime data type `TIMESTAMP WITHOUT TIME ZONE`, then *TV* is the year, month, and day <primary datetime field>s of *SV*.
- d) If *SD* is the datetime data type `TIMESTAMP WITH TIME ZONE`, then *TV* is computed by:
`CAST (CAST (SV AS TIMESTAMP WITHOUT TIME ZONE) AS DATE)`

- 14) Let *STZD* be the 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:

6.22 <cast specification>

- i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.
- ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITH TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

CAST (*TV1* AS TIME(*TSP*) WITHOUT TIME ZONE)

- iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format*.
 - iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.
- b) If *SD* is TIME WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.
 - c) If *SD* is TIME WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, computed modulo 24 hours, with implementation-defined rounding or truncation if necessary.
 - d) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined rounding or truncation if necessary.
 - e) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is:

CAST (CAST (*SV* AS TIMESTAMP(*TSP*) WITHOUT TIME ZONE) AS TIME(*TSP*) WITHOUT TIME ZONE)

- 16) If *TD* is the datetime data type TIME WITH TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

- a) If *SD* is character string, then *SV* is replaced by:

TRIM (BOTH ' ' FROM *VE*)

Case:

- i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.
- ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITHOUT TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

CAST (*TV1* AS TIME(*TSP*) WITH TIME ZONE)

- iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format*.

- iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.
- b) If *SD* is TIME WITH TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.
- c) If *SD* is TIME WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, computed modulo 24 hours, with implementation-defined rounding or truncation if necessary, and the time zone component of *TV* is *STZD*.
- d) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined rounding or truncation if necessary, and the time zone component of *TV* is the time zone displacement of *SV*.
- e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is:

CAST (CAST (SV AS TIMESTAMP(*TSP*) WITH TIME ZONE) AS TIME(*TSP*) WITH TIME ZONE)

- 17) If *TD* is the datetime data type TIMESTAMP WITHOUT TIME ZONE, then let *TSP* be the <timestamp precision> of *TD*.

Case:

- a) If *SD* is character string, then *SV* is replaced by:

TRIM (BOTH ' ' FROM VE)

Case:

- i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.
- ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITH TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

CAST (TV1 AS TIMESTAMP(*TSP*) WITHOUT TIME ZONE)
- iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format*.
- iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.
- b) If *SD* is a date, then the <primary datetime field>s hour, minute, and second of *TV* are set to 0 (zero) and the <primary datetime field>s year, month, and day of *TV* are set to their respective values in *SV*.
- c) If *SD* is TIME WITHOUT TIME ZONE, then the <primary datetime field>s year, month, and day of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and second of *TV* are set to their respective values in *SV*, with implementation-defined rounding or truncation if necessary.

6.22 <cast specification>

- d) If *SD* is TIME WITH TIME ZONE, then *TV* is:

CAST (CAST (SV AS TIMESTAMP WITH TIME ZONE)
AS TIMESTAMP WITHOUT TIME ZONE)

- e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined rounding or truncation if necessary.

- f) If *SD* is TIMESTAMP WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, with implementation-defined rounding or truncation if necessary.

- 18) If *TD* is the datetime data type TIMESTAMP WITH TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

- a) If *SD* is character string, then *SV* is replaced by:

TRIM (BOTH ' ' FROM VE)

Case:

- i) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

- ii) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITHOUT TIME ZONE, then let *TV1* be that value and let *TV* be the value of:

CAST (TV1 AS TIMESTAMP(*TSP*) WITH TIME ZONE)

- iii) If a <datetime value> does not conform to the natural rules for dates or times according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format*.

- iv) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.

- b) If *SD* is a date, then *TV* is:

CAST (CAST (SV AS TIMESTAMP(*TSP*) WITHOUT TIME ZONE)
AS TIMESTAMP(*TSP*) WITH TIME ZONE)

- c) If *SD* is TIME WITHOUT TIME ZONE, then *TC* is:

CAST (CAST (SV AS TIMESTAMP(*TSP*) WITHOUT TIME ZONE)
AS TIMESTAMP(*TSP*) WITH TIME ZONE)

- d) If *SD* is TIME WITH TIME ZONE, then the <primary datetime field>s of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and second are set to their respective values in *SV*, with implementation-defined rounding or truncation if necessary. The time zone component of *TV* is set to the time zone component of *SV*.

- e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, with a time zone displacement of *STZD*.

- f) If *SD* is **TIMESTAMP WITH TIME ZONE**, then *TV* is *SV* with implementation-defined rounding or truncation, if necessary.

19) If *TD* is interval, then

Case:

- a) If *SD* is exact numeric, then

Case:

- i) If the representation of *SV* in the data type *TD* would result in the loss of leading significant digits, then an exception condition is raised: *data exception — interval field overflow*.
- ii) Otherwise, *TV* is that representation.

- b) If *SD* is character string, then *SV* is replaced by

TRIM (BOTH ' ' FROM *VE*)

Case:

- i) If the rules for <literal> or for <unquoted interval string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.
- ii) Otherwise,
 - 1) If a <datetime value> does not conform to the natural rules for intervals according to the Gregorian calendar, then an exception condition is raised: *data exception — invalid interval format*.
 - 2) Otherwise, an exception condition is raised: *data exception — invalid datetime format*.
- c) If *SD* is interval and *TD* and *SD* have the same interval precision, then *TV* is *SV*.
- d) If *SD* is interval and *TD* and *SD* have different interval precisions, then let *Q* be the least significant <primary datetime field> of *TD*.
 - i) Let *Y* be the result of converting *SV* to a scalar in units *Q* according to the natural rules for intervals as defined in the Gregorian calendar (that is, there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year).
 - ii) Normalize *Y* to conform to the <interval qualifier> "*P TO Q*" of *TD* (again, observing the rules that there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year). Whether to truncate or round in the least significant field of the result is implementation-defined. If this would result in loss of precision of the leading datetime field of *Y*, then an exception condition is raised: *data exception — interval field overflow*.
 - iii) *TV* is the value of *Y*.

20) If *TD* is boolean, then

Case:

- a) If *SD* is character string, then *SV* is replaced by

TRIM (BOTH ' ' FROM VE)

Case:

- i) If the rules for <literal> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.
- ii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast*.
- b) If *SD* is boolean, then *TV* is *SV*.
- 21) If the <cast specification> contains a <domain name> and that <domain name> refers to a domain that contains a <domain constraint> and if *TV* does not satisfy the <check constraint> of the <domain constraint>, then an exception condition is raised: *integrity constraint violation*.

Conformance Rules

- 1) Without Feature F251, "Domain support", <cast target> shall not be a <domain name>.
- 2) Without Feature T042, "Extended LOB data type support", the declared type of <cast operand> shall not be BINARY LARGE OBJECT or CHARACTER LARGE OBJECT.
- 3) Without Feature F421, "National character", and Feature T042, "Extended LOB data type support", the declared type of <cast operand> shall not be NATIONAL CHARACTER LARGE OBJECT.
- 4) Without Feature S043, "Enhanced reference types", if the declared data type of <cast operand> is a reference type, then <cast target> shall specify a <data type> that is a reference type.

6.23 <value expression>

Function

Specify a value.

Format

```

<value expression> ::=
    <numeric value expression>
  | <string value expression>
  | <datetime value expression>
  | <interval value expression>
  | <boolean value expression>
  | <user-defined type value expression>
  | <row 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> ::=
    <value expression primary>

<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>
  | <scalar subquery>
  | <case expression>
  | <cast specification>
  | <subtype treatment>
  | <attribute or method reference>
  | <reference resolution>
  | <collection value constructor>
  | <routine invocation>
  | <field reference>
  | <element reference>
  | <method invocation>
  | <static method invocation>
  | <new specification>

<collection value constructor> ::=
    <array value expression>

```

6.23 <value expression>**Syntax Rules**

- 1) The declared type of a <value expression> is the declared type of the <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <boolean value expression>, <user-defined type value expression>, <row value expression>, <collection value expression>, or <reference value expression>, respectively.
- 2) The declared type of a <value expression primary> is the declared type of the immediately contained <unsigned value specification>, <column reference>, <set function specification>, <scalar subquery>, <case expression>, <value expression>, <cast specification>, <subtype treatment>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <field reference>, <element reference>, <method invocation>, or <static method invocation>, or the effective returns type of the immediately contained <routine invocation>, 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 <value expression primary>, which shall be a collection type.
- 6) If the declared type of a <value expression primary> is character string, then the collating sequence and coercibility characteristic of the <value expression primary> are the collating sequence and coercibility attribute of the <unsigned value specification>, <column reference>, <set function specification>, <scalar subquery>, <case expression>, <value expression>, or <cast specification> immediately contained in the <value expression primary>.
- 7) The declared type of a <collection value constructor> is the collection type of the <array value expression> that it immediately contains.
- 8) 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*.

Access Rules

None.

General Rules

- 1) The value of a <value expression> is the value of the immediately contained <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <boolean value expression>, <user-defined type value expression>, <row value expression>, <collection value expression>, or <reference value expression>.
- 2) 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.
- 3) The value of a <collection value expression> is the value of its immediately contained <value expression primary>.
- 4) The value of a <collection value constructor> is the value of the <array value constructor> that it immediately contains.

- 5) The value of a <value expression primary> is the value of the immediately contained <unsigned value specification>, <column reference>, <set function specification>, <scalar subquery>, <case expression>, <value expression>, <cast specification>, <subtype treatment>, <collection value constructor>, <field reference>, <element reference>, <method invocation>, <static method invocation>, <routine invocation>, or <attribute or method reference>.
- 6) 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”, a <value expression> shall not be a <boolean value expression>.
- 2) Without Feature S091, “Basic array support”, a <value expression> shall not specify a <collection value expression>.
- 3) Without Feature S091, “Basic array support”, a <value expression primary> shall not be a <collection value constructor>.
- 4) Without Feature S161, “Subtype treatment”, a <value expression primary> shall not be a <subtype treatment>.
- 5) Without Feature F052, “Intervals and datetime arithmetic”, a <value expression> shall not be an <interval value expression>.
- 6) Without Feature S041, “Basic reference types”, a <value expression> shall not be a <reference value expression>.

6.24 <new specification>

Function

Invoke a method on a newly-constructed value of a structured type.

Format

```
<new specification> ::=  
    NEW <routine invocation>
```

```
<new invocation> ::=  
    <method invocation>
```

Syntax Rules

- 1) Let *RN* be the <routine name> immediately contained in the <routine invocation>. Let *MN* be the <qualified identifier> immediately contained in *RN*.
- 2) Let *S* be the schema identified by the implicit or explicit <schema name> of *RN*. *S* shall include a user-defined type descriptor of an instantiable user-defined type whose user-defined type name is *RN*.
- 3) The <new specification>

NEW *RN*(*a*₁, *a*₂, ..., *a*_{*n*})
is equivalent to the <new invocation>

RN().*MN*(*a*₁, *a*₂, ..., *a*_{*n*})

Access Rules

- 1) Let *T* be the user-defined type identified by *RN*.
 - a) If <new specification> is contained in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on *T*.
 - b) Otherwise, the current privileges shall include USAGE on *T*.

General Rules

None.

Conformance Rules

- 1) Without Feature S023, “Basic structured types”, conforming Core SQL language shall not contain any <new specification>.

6.25 <subtype treatment>

Function

Modify the declared type of an expression.

Format

<subtype treatment> ::=
TREAT <left paren> <subtype operand> AS <target data type> <right paren>

<subtype operand> ::= <value expression>

<target data type> ::=
<user-defined type>

Syntax Rules

- 1) The declared type *VT* of the <value expression> shall be a structured type.
- 2) Let *DT* be the structured type identified by the <user-defined type name> simply contained in <user-defined 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) If *DT* is a proper subtype of the most specific type of *V*, then an exception condition is raised: *invalid target type specification*.
NOTE 87 – “proper subtype” and “most specific type” are defined in Subclause 4.8.3, “Subtypes and supertypes”.
- 3) The value of the result of the <subtype treatment> is *V*.

Conformance Rules

- 1) Without Feature S161, “Subtype treatment”, conforming Core SQL Language shall contain no <subtype treatment>.

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 exact numeric, 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 is 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 approximate numeric. The precision of the result is implementation-defined.
- 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.

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 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 <string value expression>

Function

Specify a character string value or a bit string value.

Format

```
<string value expression> ::=
    <character value expression>
    | <bit value expression>
    | <blob value expression>

<character value expression> ::=
    <concatenation>
    | <character factor>

<concatenation> ::=
    <character value expression> <concatenation operator> <character factor>

<character factor> ::=
    <character primary> [ <collate clause> ]

<character primary> ::=
    <value expression primary>
    | <string value function>

<blob value expression> ::=
    <blob concatenation>
    | <blob factor>

<blob factor> ::= <blob primary>

<blob primary> ::=
    <value expression primary>
    | <string value function>

<blob concatenation> ::=
    <blob value expression> <concatenation operator> <blob factor>

<bit value expression> ::=
    <bit concatenation>
    | <bit factor>

<bit concatenation> ::=
    <bit value expression> <concatenation operator> <bit factor>

<bit factor> ::= <bit primary>

<bit primary> ::=
    <value expression primary>
    | <string value function>
```


Syntax Rules

- 1) The declared type of a <character primary> shall be character string.
- 2) Character strings of different character repertoires shall not be mixed in a <character value expression>. The character repertoire of a <character value expression> is the character repertoire of its components.
- 3) Case:
 - a) If <concatenation> is specified, then:

Let $D1$ be the declared type of the <character value expression> and let $D2$ be the declared type of the <character factor>. Let M be the length in characters of $D1$ plus the length in characters of $D2$. Let VL be the implementation-defined maximum length of a variable-length character string and let FL be the implementation-defined maximum length of a fixed-length character string.

Case:

 - i) If the declared type of the <character value expression> or <character factor> is variable-length character string, then the declared type of the <concatenation> is variable-length character string with maximum length equal to the lesser of M and VL .
 - ii) If the declared type of the <character value expression> and <character factor> is fixed-length character string, then M shall not be greater than FL and the declared type of the <concatenation> is fixed-length character string with length M .
 - b) Otherwise, the declared type of the <character value expression> is the declared type of the <character factor>.
- 4) Case:
 - a) If <character factor> is specified, then

Case:

 - i) If <collate clause> is specified, then the <character value expression> has the collating sequence given in <collate clause>, and has the *Explicit* coercibility characteristic.
 - ii) Otherwise, if <value expression primary> or <string value function> are specified, then the collating sequence and coercibility characteristic of the <character factor> are specified in Subclause 6.3, "<value specification> and <target specification>", and Subclause 6.18, "<string value function>", respectively.
 - b) If <concatenation> is specified, then the collating sequence and the coercibility characteristic are determined as specified for dyadic operators in Subclause 4.2.3, "Rules determining collating sequence usage".
- 5) The declared type of <blob primary> shall be binary string.
- 6) If <blob concatenation> is specified, then let M be the length in octets of the <blob value expression> plus the length in octets of the <blob factor> and let VL be the implementation-defined maximum length of a binary string. The declared type of <blob concatenation> is binary string with maximum length equal to the lesser of M and VL .

- 7) The declared type of a <bit primary> shall be bit string.
- 8) Case:
- a) If <bit concatenation> is specified, then let $D1$ be the declared type of the <bit value expression>, let $D2$ be the declared type of the <bit factor>, let M be the length in bits of $D1$ plus the length in bits of $D2$, let VL be the implementation-defined maximum length of a variable-length bit string, and let FL be the implementation-defined maximum length of a fixed-length bit string.
- Case:
- i) If the declared type of the <bit value expression> or <bit factor> is variable-length bit string, then the declared type of the <bit concatenation> is variable-length bit string with maximum length equal to the lesser of M and VL .
- ii) If the declared type of the <bit value expression> and <bit factor> is fixed-length bit string, then M shall not be greater than FL and the declared type of the <bit concatenation> is fixed-length bit string with length M .
- iii) Otherwise, the declared type of a <bit value expression> is the declared type of the <bit factor>.

Access Rules

None.

General Rules

- 1) If the value of any <character primary>, <bit primary>, or <blob primary> simply contained in a <string value expression> is the null value, then the result of the <string value expression> is the null value.
- 2) If <concatenation> is specified, then let $S1$ and $S2$ be the result of the <character value expression> and <character factor>, respectively.

Case:

- a) If either $S1$ or $S2$ is the null value, then the result of the <concatenation> is the null value.
- b) Otherwise, let S be the string consisting of $S1$ followed by $S2$ and let M be the length of S .

Case:

- i) If the most specific type of either $S1$ or $S2$ is variable-length character string, then

Case:

- 1) If M is less than or equal to VL , then the result of the <concatenation> is S with length M .
- 2) 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 .

- 3) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.
- ii) 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 <bit concatenation> is specified, then let *S1* and *S2* be the result of the <bit value expression> and <bit factor>, respectively.

Case:

- a) If either *S1* or *S2* is the null value, then the result of the <bit 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 bits of *S*.

Case:

- i) If the most specific type of either *S1* or *S2* is variable-length bit string, then

Case:

- 1) If *M* is less than or equal to *VL*, then the result of the <bit concatenation> is *S* with length *M*.
- 2) If *M* is greater than *VL* and the right-most *M*–*VL* bits of *S* are all 0-valued, then the result of the <bit concatenation> is the first *VL* bits of *S* with length *VL*.
- 3) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

- ii) If the most specific types of both *S1* and *S2* are fixed-length bit string, then the result of the <bit concatenation> is *S*.
- 4) If <blob concatenation> is specified, then let *S1* and *S2* be the result of the <blob value expression> and <blob factor>, respectively.

Case:

- a) If either *S1* or *S2* is the null value, then the result of the <blob concatenation> is the null value.
- b) Otherwise, let *S* be the string consisting of *S1* followed by *S2* and let *M* be the length in octets of *S*.

Case:

- i) If *M* is less or equal to *VL*, then the result of the <blob concatenation> is *S* with length *M*.
- ii) If *M* is greater than *VL* and the right-most *M*–*VL* octets of *S* are all X'00', then the result of the <blob concatenation> is the first *VL* octets of *S* with length *VL*.
- iii) Otherwise, an exception condition is raised: *data exception — string data, right truncation*.

- 5) 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

- 1) Without Feature T042, “Extended LOB data type support”, neither operand of <concatenation> shall be of declared type CHARACTER LARGE OBJECT.
- 2) Without Feature T042, “Extended LOB data type support”, neither operand of <blob concatenation> shall be of declared type BINARY LARGE OBJECT.
- 3) Without Feature F421, “National character”, and Feature T042, “Extended LOB data type support”, neither operand of <concatenation> shall be of declared type NATIONAL CHARACTER LARGE OBJECT.
- 4) Without Feature F691, “Collation and translation”, conforming SQL language shall not contain any <collate clause>.
- 5) Without Feature F511, “BIT data type”, conforming SQL language shall contain no <bit value expression>.

6.28 <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>.

6.28 <datetime value expression>

- 5) If the <datetime value expression> immediately contains either <plus sign> or <minus sign>, then:
 - a) The <interval value expression> or <interval term> shall contain only <primary datetime field>s that are contained within the <datetime value expression> or <datetime term>.
 - b) The result of the <datetime value expression> contains the same <primary datetime field>s that are contained in the <datetime value expression> or <datetime term>, with a fractional seconds precision that is the greater of the fractional seconds precisions, if any, of either the <datetime value expression> and <interval term>, or the <datetime term> and <interval value expression> that it simply contains.
- 6) The declared type of the <interval primary> immediately contained in a <time zone specifier> shall be INTERVAL HOUR TO MINUTE.

Access Rules

None.

General Rules

- 1) If the value of any <datetime primary>, <interval value expression>, <datetime value expression>, or <interval term> simply contained in a <datetime value expression> is the null value, then the result of the <datetime value expression> is the null value.
- 2) If <time zone> is specified and the <interval primary> immediately contained in <time zone specifier> is null, then the result of the <datetime value expression> is the null value.
- 3) The value of a <datetime primary> is the value of the immediately contained <value expression primary> or <datetime value function>.
- 4) In the following General Rules, arithmetic is performed so as to maintain the integrity of the datetime data type that is the result of the <datetime term> or <datetime value expression>. This may involve carry from or to the immediately next more significant <primary datetime field>. If the data type of the <datetime term> or <datetime value expression> is time with or without time zone, then arithmetic on the HOUR <primary datetime field> is undertaken modulo 24. If the <interval value expression> or <interval term> is a year-month interval, then the DAY field of the result is the same as the DAY field of the <datetime term> or <datetime value expression>.
- 5) The value of a <datetime term> is determined as follows. Let *DT* be the declared type, *DV* the UTC component of the value, and *TZD* the time zone component, if any, of the <datetime primary> simply contained in the <datetime term>, and let *STZD* be the current default time zone displacement of the SQL-session.

Case:

- a) If <time zone> is not specified, then the value of <datetime term> is *DV*.
- b) Otherwise:
 - i) Case:
 - 1) If *DT* is datetime with time zone, then the UTC component of the <datetime term> is *DV*.

- 2) Otherwise, the UTC component of the <datetime term> is $DV - STZD$.
 - ii) Case:
 - 1) If LOCAL is specified, then let TZ be $STZD$.
 - 2) If TIME ZONE is specified, then, if the value of the <interval primary> immediately contained in <time zone specifier> is less than INTERVAL - '12:59' or greater than INTERVAL + '13:00', 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 UTC component of the result 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 88 – For the permissible range of values for <primary datetime field>s, see Table 11, "Valid values for datetime fields".

Conformance Rules

- 1) Without Feature F052, "Intervals and datetime arithmetic", <datetime value expression> shall not specify <plus sign> or <minus sign>.
- 2) Without Feature F411, "Time zone specification", <datetime factor> shall not specify <time zone>.

6.29 <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 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.
 - 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.

NOTE 89 – Interval fields are effectively defined by Table 6, “Fields in year-month INTERVAL values”, and Table 7, “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.

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>, then the value of IP is the value of VEP .
 - b) 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

$$\text{CAST} (\text{CAST} (X \text{ AS INTERVAL } Q) \text{ AS } E1)$$
 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

$$\text{CAST} ((E \text{ OP } Y) \text{ AS INTERVAL } Q).$$

- 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 any <interval value expression>.

6.30 <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>  
    | <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 <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:
$$(\text{NOT} (BP \text{ IS } TV))$$
- 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 expression> that simply contains a <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 <value expression primary>, then
Case:
 - i) If *BVE* is of the form “<left paren> <value expression> <right paren>” and the <value expression> is a known-not-null condition for *X*, then *BVE* is a *known-not-null condition* for *X*.
 - ii) Otherwise, *BVE* is not a known-not-null condition for *X*.
- c) 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*.
- d) 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 90 – 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))”
- e) 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 a *known-not-null condition* for *X*.
 - ii) Otherwise, *BVE* is not a known-not-null condition for *X*.
- f) If *BVE* is of the form “*BVE1* OR *BVE2*”, then *BVE* is not a known-not-null condition for *X*.
NOTE 91 – 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.

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 13, “Truth table for the AND boolean operator”, Table 14, “Truth table for the OR boolean operator”, and Table 15, “Truth table for the IS boolean operator” specify the semantics of AND, OR, and IS, respectively.

Table 13—Truth table for the AND boolean operator

AND	<u>true</u>	<u>false</u>	<u>unknown</u>
<u>true</u>	<u>true</u>	<u>false</u>	<u>unknown</u>
<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>
<u>unknown</u>	<u>unknown</u>	<u>false</u>	<u>unknown</u>

Table 14—Truth table for the OR boolean operator

OR	<u>true</u>	<u>false</u>	<u>unknown</u>
<u>true</u>	<u>true</u>	<u>true</u>	<u>true</u>
<u>false</u>	<u>true</u>	<u>false</u>	<u>unknown</u>
<u>unknown</u>	<u>true</u>	<u>unknown</u>	<u>unknown</u>

Table 15—Truth table for the IS boolean operator

IS	TRUE	FALSE	UNKNOWN
<u>true</u>	<u>true</u>	<u>false</u>	<u>false</u>
<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>
<u>unknown</u>	<u>false</u>	<u>false</u>	<u>true</u>

Conformance Rules

- 1) Without Feature T031, “BOOLEAN data type”, a <boolean primary> shall not specify a <non-parenthesized value expression primary>.
- 2) Without Feature F571, “Truth value tests”, a <boolean test> shall not specify a <truth value>.

6.31 <array value expression>

Function

Specify an array value.

Format

```

<array value expression> ::=
    <array value constructor>
    | <array concatenation>
    | <value expression primary>

```

```

<array concatenation> ::=
    <array value expression 1> <concatenation operator> <array value expression 2>

```

```

<array value expression 1> ::= <array value expression>

```

```

<array value expression 2> ::= <array value expression>

```

Syntax Rules

- 1) The declared type of <value expression primary> shall be an array type.
- 2) The declared type of the <array value expression> is the declared type of the immediately contained <array value constructor>, <array concatenation>, or <value expression primary>.
- 3) If <array concatenation> is specified, then:
 - a) Let *DT* be the data type determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of <array value expression 1> and <array value expression 2>.
 - 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 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 value expression 2>.

Access Rules

None.

General Rules

- 1) The value of the result of <array value expression> is the value of the immediately contained <array value constructor>, <array concatenation>, or <value expression primary>.
- 2) If <array concatenation> is specified, then:
 - a) Let *AV1* be the value of <array value expression 1> and let *AV2* be the value of <array value expression 2>.

6.31 <array value expression>

- b) If either *AV1* or *AV2* is the null value, then the result of the <array concatenate function> is the null value.
- 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 any <array value expression>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

6.32 <array value constructor>

Function

Specify construction of an array.

Format

<array value constructor> ::=
 <array value list constructor>

<array value list constructor> ::=
 ARRAY <left bracket or trigraph> <array element list> <right bracket or trigraph>

<array element list> ::=
 <array element> [{ <comma> <array element> }...]

<array element> ::=
 <value expression>

Syntax Rules

- 1) The declared type of the <array value constructor> is an array type with element declared type *DT*, where *DT* is the declared type determined by applying Subclause 9.3, “Data types of results of aggregations”, to the declared types of the <array element>s immediately contained in the <array element list> of this <array value constructor>.

Access Rules

None.

General Rules

- 1) The result of <array value constructor> 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*.

Conformance Rules

- 1) Without Feature S091, “Basic array support”, conforming SQL language shall not contain any <array value constructor>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

7 Query expressions

7.1 <row value constructor>

Function

Specify a value or list of values to be constructed into a row or partial row.

Format

```

<row value constructor> ::=
    <row value constructor element>
  | [ 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> ::=
    <contextually typed row value constructor element>
  | [ 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>

```

Syntax Rules

- 1) A <row value constructor element> immediately contained in a <row value constructor> shall not be a <value expression> of the form “<left paren> <value expression> <right paren>”.
NOTE 92 – This Rule removes a syntactic ambiguity. A <row value constructor> of this form is permitted, but is parsed in the form “<left paren> <row value constructor element list> <right paren>”.
- 2) A <row value constructor element> immediately contained in a <row value constructor> shall not be a <value expression> that is a <row value expression>.
NOTE 93 – This Rule removes a syntactic ambiguity, since otherwise a <row value constructor> could be a <row value expression>, and a <row value expression> could be a <row value constructor>.
- 3) Let *RVC* be the <row value constructor>.

7.1 <row value constructor>

Case:

- a) If *RVC* immediately contains a <row subquery>, then the declared type of *RVC* is the declared type of that <subquery>.
 - b) Otherwise, the declared type of *RVC* 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 *RVC*. The data type is the declared type of *X* and the <field name> is implementation-dependent and not equivalent to the <column name> name of any column or field, other than itself, of a table referenced by any <table reference> contained in the SQL-statement.
- 4) The degree of a <row value constructor> 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 the <row value constructor> immediately contains a <row value constructor element> *X*, then the result of the <row value constructor> is a row containing a single column whose value is the value of *X*.
 - b) If a <row value constructor element list> is specified, then the result of the <row value constructor> is a row of columns, the value of whose *i*-th column is the value of the *i*-th <row value constructor element> in the <row value constructor element list>.
 - c) If the <row value constructor> is a <row subquery>, then:
 - i) Let *R* be the result of the <row subquery> and let *D* be the degree of *R*.
 - ii) If the cardinality of *R* is 0 (zero), then the result of the <row value constructor> is *D* null values.
 - iii) If the cardinality of *R* is 1 (one), then the result of the <row value constructor> is *R*.

Conformance Rules

- 1) Without Feature T051, "Row types", ROW shall not be specified.
- 2) Without Feature S091, "Basic array support", <empty specification> shall not be specified.
- 3) Without Feature F641, "Row and table constructors", a <row value constructor> that is not simply contained in a <table value constructor> shall not contain more than one <row value constructor element>.

- 4) Without Feature F641, “Row and table constructors”, a <row value constructor> shall not be a <row subquery>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

7.2 <row value expression>

Function

Specify a row value.

Format

```
<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 special case> ::=  
    <value specification>  
    | <value expression>
```

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 <row value constructor>.
- 3) 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>.

Access Rules

None.

General Rules

- 1) A <row value special case> specifies the row value denoted by the <value specification> or <value expression>.
- 2) A <row value expression> specifies the row value denoted by the <row value special case> or <row value constructor>.
- 3) A <contextually typed row value expression> specifies the row value denoted by the <row value special case> or <contextually typed row value constructor>.

Conformance Rules

- 1) Without Feature T051, "Row types", conforming SQL language shall not contain any <row value expression> or <contextually typed row value expression> that immediately contains <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> ::=
    <row value expression> [ { <comma> <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 <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 contains a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.
- 4) Let *TVC* be some <table value constructor> consisting of n <row value expression>s or some <contextually typed table value constructor> consisting of n <contextually typed row value expression>s. Let RVE_i , $1 \text{ (one)} \leq i \leq n$, denote the i -th <row value expression> or the i -th <contextually typed row value expression>. The row type of *TVC* is determined by applying Subclause 9.3, "Data types of results of aggregations", to the row types RVE_i , $1 \text{ (one)} \leq i \leq n$.

Access Rules

None.

General Rules

- 1) If the result of any <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 <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”, the <row value expression list> of a <table value constructor> shall contain exactly one <row value constructor> *RVE*. *RVE* shall be of the form “(<row value constructor element list>)”.
- 2) Without Feature F641, “Row and table constructors”, conforming SQL language shall not contain any <table value constructor>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

7.4 <table expression>

Function

Specify a table or a grouped table.

Format

```
<table expression> ::=  
    <from clause>  
    [ <where clause> ]  
    [ <group by clause> ]  
    [ <having 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 the last of the immediately contained clauses specified in the <table expression>.
- 2) A <table expression> is *possibly non-deterministic* if it contains a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.
- 3) 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 in $TRLPR$, let $TRLLR_i$ be the corresponding evaluation of $TRLL$ under all outer references contained in $TRLL$. Let $SUBR_i$ be the table containing every row formed by concatenating R_i with some row of $TRLLR_i$. Every row RR in $SUBR_i$ is a row in $TRLR$, and the number of occurrences of RR in $TRLR$ is the sum of the numbers of occurrences of RR in every occurrence of $SUBR_i$.

The result of the <table reference list> is $TRLR$ with the columns reordered according to the ordering of the descriptors of the columns of the <table reference list>.

- 2) The result of the <from clause> is $TRLR$.

Conformance Rules

None.

7.6 <table reference>

Function

Reference a table.

Format

```
<table reference> ::=
    <table primary>
    | <joined table>

<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> ]
    | <only spec>
        [ [ AS ] <correlation name>
            [ <left paren> <derived column list> <right paren> ] ]
    | <left paren> <joined table> <right paren>

<only spec> ::=
    ONLY <left paren> <table or query name> <right paren>

<lateral derived table> ::=
    LATERAL <left paren> <query expression> <right paren>

<collection derived table> ::=
    UNNEST <left paren> <collection value expression> <right paren>
    [ WITH ORDINALITY ]

<derived table> ::= <table subquery>

<table or query name> ::=
    <table name>
    | <query name>

<derived column list> ::= <column name list>

<column name list> ::=
    <column name> [ { <comma> <column name> }... ]
```

Syntax Rules

- 1) If a <table reference> *TR* specifies a <collection derived table> *CDT*, then let *C* be the <collection value expression> immediately contained in *CDT*, let *CN* be the <correlation name> immediately contained in *TR*, and let *TEMP* be an <identifier> that is not equivalent to *CN* nor to any other <identifier> contained in *TR*.
 - a) Case:
 - i) If *TR* specifies a <derived column list> *DCL*, then

Case:

- 1) If *CDT* specifies *WITH ORDINALITY*, then *DCL* shall contain 2 <column name>s. Let *N1* and *N2* be respectively the first and second of those <column name>s.
 - 2) Otherwise, *DCL* shall contain 1 (one) <column name>; let *N1* be that <column name>. Let *N2* be a <column name> that is not equivalent to *N1*, *CN*, *TEMP*, or any other <identifier> contained in *TR*.
- ii) Otherwise, 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 *TR*.
- b) 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)
)
```

c) Case:

- i) If *TR* specifies a <derived column list> *DCL*, then let *PDCLP* be

(*DCL*)

- ii) Otherwise, let *PDCLP* be a zero-length string.

d) Case:

- i) If *CDT* specifies *WITH ORDINALITY*, then let *ELDT* be:

LATERAL (*RECQP* SELECT * FROM TEMP AS *CN* *PDCLP*)

- ii) Otherwise, let *ELDT* be:

LATERAL (*RECQP* SELECT *N1* FROM TEMP AS *CN* *PDCLP*)

e) *CDT* is equivalent to the <lateral derived table> *ELDT*.

- 2) A <correlation name> immediately contained in a <table reference> *TR* is *exposed* by *TR*. A <table or query name> immediately contained in a <table reference> *TR* is *exposed* by *TR* if and only if *TR* does not specify a <correlation name>.

3) Case:

- a) If a <table reference> *TR* is contained in a <from clause> *FC* with no intervening <derived table>, then the *scope clause* *SC* of *TR* is the <select statement: single row> or innermost

7.6 <table reference>

<query specification> that contains *FC*. The scope of the exposed <correlation name> or exposed <table or query name> of *TR* is the <select list>, <where clause>, <group by clause>, and <having 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*.

- b) Otherwise, the *scope clause SC* of *TR* is the outermost <joined table> that contains *TR* with no intervening <derived table>. The scope of the exposed <correlation name> or exposed <table or query name> of *TR* is the <join condition> of *SC* and of all <joined table>s contained in *SC* that contain *TR*.
- 4) A <table or query name> that is exposed by a <table reference> *TR* shall not be the same as any other <table or query name> that is exposed by a <table reference> with the same scope clause as *TR*.
- 5) A <correlation name> that is exposed by a <table reference> *TR* shall not be the same as any other <correlation name> that is exposed by a <table reference> with the same scope clause as *TR* and shall not be the same as the <qualified identifier> of any <table or query name> that is exposed by a <table reference> with the same scope clause as *TR*.
- 6) A <table or query name> immediately contained in a <table reference> *TR* has a scope clause and scope defined by that <table reference> if and only if the <table or query name> is exposed by *TR*.
- 7) If *TR* immediately 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*.
- 8) No <column name> shall be specified more than once in a <derived column list>.
- 9) If a <derived column list> is specified in a <table reference>, 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> of that <table reference>, and the name of the *i*-th column of that <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>.
- 10) The row type of a <lateral derived table> is the row type of the immediately contained <query expression>.
- 11) Case:
 - a) If no <derived column list> is specified, then the row type *RT* of the <table reference> is the row type of its immediately contained <table or query name>, <derived table>, <lateral derived table>, or <joined table>.
 - b) Otherwise, the row type *RT* of the <table reference> 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>, or <lateral derived table>, of the table identified by the <table or query name> immediately contained in the <table reference>.

- 12) A <derived table> or <lateral derived table> is an *updatable derived table* if and only if the <query expression> simply contained in the <subquery> of the <table subquery> of the <derived table> is updatable.
- 13) A <derived table> or <lateral derived table> is an *insertable-into derived table* if and only if the <query expression> simply contained in the <subquery> of the <table subquery> of the <derived table> is insertable-into.
- 14) A <collection derived table> is not updatable.
- 15) Case:
 - a) If *TR* simply contains a <table name> that identifies a base table, then every column of the table identified by *TR* is called an *updatable column* of *TR*.
 - b) If *TR* simply contains a <table name> that identifies a view, then every updatable column of the view identified by *TR* is called an *updatable column* of *TR*.
 - c) If *TR* immediately 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> is called an *updatable column* of *TR*.
- 16) If the <table or query name> immediately contained in <table reference> is not a query name in scope, then let *T* be the table identified by the <table name> immediately contained in <table or query name>. If the <table reference> is not contained in a <schema definition>, then the schema identified by the explicit or implicit qualifier of the <table name> shall include the descriptor of *T*. If the <table reference> is contained in a <schema definition> *S*, then the schema identified by the explicit or implicit qualifier of the <table name> shall include the descriptor of *T*, or *S* shall contain a <schema element> that creates the descriptor of *T*.
NOTE 94 – “*query name in scope*” is defined in Subclause 7.12, “<query expression>”.

Access Rules

- 1) If <table reference> immediately contains a <table or query name> that is a <table name>, then:
 - a) Let *T* be the table identified by the <table name> immediately contained in the <table or query name> immediately contained in <table reference>.
 - b) If *T* is a base table or a viewed table and the <table reference> is contained in any of:
 - A <query expression> simply contained in a <cursor specification>, a <view definition>, or an <insert statement>.
 - A <table expression> or <select list> immediately contained in a <select statement: single row>.
 - A <search condition> immediately contained in a <delete statement: searched> or an <update statement: searched>.
 - A <value expression> simply contained in a <row value expression> immediately contained in a <set clause>.
- then
- Case:

7.6 <table reference>

- i) If <table reference> is contained 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*.
- ii) Otherwise, the current privileges shall include SELECT on at least one column of *T*.
NOTE 95 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.
- c) If the <table reference> is contained in a <query expression> simply contained in a <view definition> then the applicable privileges of the <authorization identifier> that owns the view shall include SELECT for at least one column of *T*.
- d) If *TR* immediately contains <only spec>, then
Case:
 - i) If <table reference> is contained in a <schema definition>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.
 - ii) Otherwise, the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.
NOTE 96 – “applicable privileges” and *current privileges* are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) A <correlation name> or exposed <table or query name> contained in a <table reference> defines that <correlation name> or <table or query name> to be an identifier of the table identified by the <table or query name> or <derived table> of that <table reference>.
- 2) If the <table reference> simply contains a <table name> *TN* and *TN* identifies a view or a base table *T*, then
Case:
 - a) If ONLY is specified, then the <table reference> references the table that consists of every row in *T*, except those rows that have a subrow in a proper subtable of *T*.
 - b) Otherwise, the <table reference> references the table that consists of every row of *T*.
- 3) If a <lateral derived table> *LDT* immediately containing <query expression> *QE* is specified, then the result of *LDT* is the result of *QE*.

Conformance Rules

- 1) Without Feature S091, “Basic array support”, a <table reference> shall not contain a <collection derived table>.
- 2) Without Feature T491, “LATERAL derived table”, conforming SQL language shall not specify a <lateral derived table>.
- 3) Without Feature T121, “WITH (excluding RECURSIVE) in query expression”, a <table reference> shall not contain a <query name>.
- 4) Without Feature S111, “ONLY in query expressions”, a <table reference> shall not specify ONLY.

- 5) Without Feature F591, “Derived tables”, conforming SQL language shall not specify a <derived table>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

7.7 <joined table>

Function

Specify a table derived from a Cartesian product, inner or outer join, or union join.

Format

```
<joined table> ::=
    <cross join>
  | <qualified join>
  | <natural join>
  | <union join>

<cross join> ::=
    <table reference> CROSS JOIN <table primary>

<qualified join> ::=
    <table reference> [ <join type> ] JOIN <table reference>
    <join specification>

<natural join> ::=
    <table reference> NATURAL [ <join type> ] JOIN <table primary>

<union join> ::=
    <table reference> UNION JOIN <table primary>

<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) No <column name> contained in a <column name list> shall identify a column whose declared type is a large object string or an array type.
- 2) Let TR_1 and TR_2 be the first and second <table reference>s of the <joined table>, respectively. Let T_1 and T_2 be the tables identified by TR_1 and TR_2 , respectively. Let TA and TB be the correlation names of TR_1 and TR_2 , respectively. Let CP be:

SELECT * FROM TR_1 , TR_2

- 3) A <joined table> is *possibly non-deterministic* if either T_1 or T_2 is possibly non-deterministic.
- 4) If a <qualified join> or <natural join> is specified and a <join type> is not specified, then INNER is implicit.
- 5) If a <qualified join> containing a <join condition> is specified, then:

- a) Each column reference directly contained in the <search condition> shall unambiguously reference a column of T_1 or T_2 or be an outer reference.
- b) If 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> and the <set function specification> shall contain a column reference that is an outer reference.

NOTE 97 – *Outer reference* is defined in Subclause 6.6, "<column reference>".

- 6) If neither NATURAL is specified nor a <join specification> immediately containing a <named columns join> is specified, then the descriptors of the columns of the result of the <joined table> are the same as the descriptors of the columns of CP , with the possible exception of the nullability characteristics of the columns.
- 7) If NATURAL is specified or if a <join specification> immediately containing a <named columns join> is specified, then:
 - a) If NATURAL is specified, then let *common column name* be a <column name> that is the <column name> of exactly one column of T_1 and the <column name> of exactly one column of T_2 . T_1 shall not have any duplicate common column names and T_2 shall not have any duplicate common column names. Let *corresponding join columns* refer to all columns of T_1 and T_2 that have common column names, if any.
 - b) If a <named columns join> is specified, then every <column name> in the <join column list> shall be the <column name> of exactly one column of T_1 and the <column name> of exactly one column of T_2 . Let *common column name* be the name of such a column. Let *corresponding join columns* refer to the columns of T_1 and T_2 identified in the <join column list>.
 - c) Let C_1 and C_2 be a pair of corresponding join columns contained in T_1 and T_2 , respectively. C_1 and C_2 shall be comparable.
 - d) If there is at least one corresponding join column, then let $SLCC$ be a <select list> of <derived column>s of the form

COALESCE ($TA.C$, $TB.C$) AS C

for every column C that is a corresponding join column, taken in order of their ordinal positions in T_1 .

- e) If T_1 contains at least one column that is not a corresponding join column, then let SLT_1 be a <select list> of <derived column>s of the form

$TA.C$

for every column C of T_1 that is not a corresponding join column, taken in order of their ordinal positions in T_1 .

7.7 <joined table>

- f) If T_2 contains at least one column that is not a corresponding join column, then let SLT_2 be a <select list> of <derived column>s of the form

$TB.C$

for every column C of T_2 that is not a corresponding join column, taken in order of their ordinal positions in T_2 .

- g) Let the <select list> SL be defined as

Case:

- i) If all of the columns of T_1 and T_2 are corresponding join columns, then let SL be " $SLCC$ ".
- ii) If T_1 contains no corresponding join columns and T_2 contains no corresponding join columns, then let SL be " SLT_1, SLT_2 ".
- iii) If T_1 contains no columns other than corresponding join columns, then let SL be " $SLCC, SLT_2$ ".
- iv) If T_2 contains no columns other than corresponding join columns, then let SL be " $SLCC, SLT_1$ ".
- v) Otherwise, let SL be " $SLCC, SLT_1, SLT_2$ ".

The descriptors of the columns of the result of the <joined table>, with the possible exception of the nullability characteristics of the columns, are the same as the descriptors of the columns of the result of

$SELECT SL FROM TR_1, TR_2$

- 8) 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.
- 9) For every column CR of the result of the <joined table> that is not a corresponding join column and that corresponds to a column C_1 of T_1 , CR is *possibly nullable* if any of the following conditions are true:
 - a) RIGHT, FULL, or UNION is specified, or
 - b) INNER, LEFT, or CROSS JOIN is specified or implicit and C_1 is possibly nullable.
- 10) For every column CR of the result of the <joined table> that is not a corresponding join column and that corresponds to a column C_2 of T_2 , CR is *possibly nullable* if any of the following conditions are true:
 - a) LEFT, FULL, or UNION is specified, or
 - b) INNER, RIGHT, or CROSS JOIN is specified or implicit and C_2 is possibly nullable.
- 11) For every column CR of the result of the <joined table> that is a corresponding join column and that corresponds to a column C_1 of T_1 and C_2 of T_2 , CR is *possibly nullable* if any of the following conditions are true:
 - a) LEFT or FULL is specified and C_1 is possibly nullable, or

- b) RIGHT or FULL is specified and C_2 is possibly nullable.

Access Rules

None.

General Rules

1) Case:

- a) If <union join> is specified, then let T be the empty set.
- b) If a <cross join> is specified, then let T be the multiset of rows of CP .
- c) If a <join condition> is specified, then let T be the multiset of rows of CP for which the specified <search condition> is true.
- d) If NATURAL is specified or <named columns join> is specified, then

Case:

- i) If there are corresponding join columns, then let T be the multiset of rows of CP for which the corresponding join columns have equal values.
 - ii) Otherwise, let T be the multiset of rows of CP .
- 2) Let P_1 be the multiset of rows of T_1 for which there exists in T some row that is the concatenation of some row R_1 of T_1 and some row R_2 of T_2 . Let P_2 be the multiset of rows of T_2 for which there exists in T some row that is the concatenation of some row R_1 of T_1 and some row R_2 of T_2 .
 - 3) Let U_1 be those rows of T_1 that are not in P_1 and let U_2 be those rows of T_2 that are not in P_2 .
 - 4) Let D_1 and D_2 be the degree of T_1 and T_2 , respectively. Let X_1 be U_1 extended on the right with D_2 columns containing the null value. Let X_2 be U_2 extended on the left with D_1 columns containing the null value.
 - 5) Let XN_1 and XN_2 be effective distinct names for X_1 and X_2 , respectively. Let TN be an effective name for T .

Case:

- a) If INNER or <cross join> is specified, then let S be the multiset of rows of T .
- b) If LEFT is specified, then let S be the multiset of rows resulting from:


```
SELECT * FROM T
UNION ALL
SELECT * FROM X1
```
- c) If RIGHT is specified, then let S be the multiset of rows resulting from:


```
SELECT * FROM T
UNION ALL
SELECT * FROM X2
```

7.7 <joined table>

- d) If FULL is specified, then let S be the multiset of rows resulting from:

```
SELECT * FROM T
UNION ALL
SELECT * FROM X1
UNION ALL
SELECT * FROM X2
```

- e) If UNION is specified, then let S be the multiset of rows resulting from:

```
SELECT * FROM X1
UNION ALL
SELECT * FROM X2
```

- 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 column of T_1 if i is less than or equal to D_1 . Column CS_j of S corresponds to the $(j-D_1)$ -th column of T_2 for j greater than D_1 .

- ii) If there is at least one corresponding join column, then let $SLCC$ be a <select list> of derived columns of the form

COALESCE (CS_i , CS_j)

for every pair of columns CS_i and CS_j , where CS_i and CS_j correspond to columns of T_1 and T_2 that are a pair of corresponding join columns.

- iii) If T_1 contains one or more columns that are not corresponding join columns, then let SLT_1 be a <select list> of the form:

CS_i

for every column CS_i of S that corresponds to a column of T_1 that is not a corresponding join column, taken in order of their ordinal position in S .

- iv) If T_2 contains one or more columns that are not corresponding join columns, then let SLT_2 be a <select list> of the form:

CS_j

for every column CS_j of S that corresponds to a column of T_2 that is not a corresponding join column, taken in order of their ordinal position in S .

- v) Let the <select list> SL be defined as:

- 1) If all the columns of T_1 and T_2 are corresponding join columns, then let SL be

$SLCC$

- 2) If T_1 contains no corresponding join columns and T_2 contains no corresponding join columns, then let SL be

SLT_1 , SLT_2

- 3) If T_1 contains no columns other than corresponding join columns, then let SL be

$SLCC$, SLT_2

- 4) If T_2 contains no columns other than corresponding join columns, then let SL be
 $SLCC, SLT_1$
- 5) Otherwise, let SL be
 $SLCC, SLT_1, SLT_2$
- 6) The result of the <joined table> is the multiset of rows resulting from:
 $SELECT SL FROM SN$
- vi) Otherwise, the result of the <joined table> is S .

Conformance Rules

- 1) Without Feature F401, "Extended joined table", conforming SQL language shall contain no <cross join>.
- 2) Without Feature F401, "Extended joined table", conforming SQL language shall not specify UNION JOIN.
- 3) Without Feature F401, "Extended joined table", conforming SQL language shall not specify NATURAL.
- 4) Without Feature F401, "Extended joined table", conforming SQL language shall not specify FULL.

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) Let *T* be the result of the preceding <from clause>. Each column reference directly contained in the <search condition> shall unambiguously reference a column of *T* or be an outer reference.
NOTE 98 – *Outer reference* is defined in Subclause 6.6, “<column reference>”.
- 2) If the <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> and every column reference contained in the <set function specification> shall be an outer reference.

NOTE 99 – *Outer reference* is defined in Subclause 6.6, “<column reference>”.

- 3) No column reference contained in a <subquery> in the <search condition> that references a column of *T* shall be specified in a <set function specification>.

Access Rules

None.

General Rules

- 1) The <search condition> is applied to each row of *T*. The result of the <where clause> is a table of those rows of *T* for which the result of the <search condition> is true.
- 2) Each <subquery> in the <search condition> is effectively executed for each row of *T* and the results used in the application of the <search condition> to the given row of *T*. If any executed <subquery> contains an outer reference to a column of *T*, then the reference is to the value of that column in the given row of *T*.

NOTE 100 – *Outer reference* is defined in Subclause 6.6, “<column reference>”.

Conformance Rules

- 1) Without Feature F441, “Extended set function support”, a <value expression> directly contained in the <search condition> shall not contain 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 <grouping specification>

<grouping specification> ::=
    <grouping column reference>
    | <rollup list>
    | <cube list>
    | <grouping sets list>
    | <grand total>
    | <concatenated grouping>

<rollup list> ::=
    ROLLUP <left paren> <grouping column reference list> <right paren>

<cube list> ::=
    CUBE <left paren> <grouping column reference list> <right paren>

<grouping sets list> ::=
    GROUPING SETS <left paren> <grouping set list> <right paren>

<grouping set list> ::=
    <grouping set> [ { <comma> <grouping set> }... ]

<concatenated grouping> ::=
    <grouping set> <comma> <grouping set list>

<grouping set> ::=
    <ordinary grouping set>
    | <rollup list>
    | <cube list>
    | <grand total>

<ordinary grouping set> ::=
    <grouping column reference>
    | <left paren> <grouping column reference list> <right paren>

<grand total> ::= <left paren> <right paren>

<grouping column reference list> ::=
    <grouping column reference> [ { <comma> <grouping column reference> }... ]

<grouping column reference> ::=
    <column reference> [ <collate clause> ]

```

7.9 <group by clause>

Syntax Rules

- 1) Each <column reference> in the <group by clause> 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 101 – “Column reference” is defined in Subclause 6.6, “<column reference>”.
- 2) If the declared type of a grouping column is a user-defined type *DT*, then the comparison form of *DT* shall be FULL.
- 3) Let *QS* be the <query specification> that simply contains the <group by clause>, and let *SL*, *FC*, *WC*, and *HC* be the <select list>, the <from clause>, the <where clause> if any, and the <having clause> if any, respectively, that are simply contained in *QS*.
- 4) Let *SING* be the <select list> constructed by removing from *SL* every <select sublist> that is not a <derived column> that contains at least one <set function specification>.
- 5) The declared type of a grouping column shall not be large object string, an array type, or a distinct type whose source type is large object string or an array type.
- 6) For every grouping column, if <collate clause> is specified, then the declared type of the column reference shall be character string. The column descriptor of the corresponding column in the result has the collating sequence specified in <collate clause> and the coercibility characteristic *Explicit*.
- 7) If <grouping specification> immediately contains <rollup list>, then let GCR_i range over the n <grouping column reference>s contained in the <rollup list>. *QS* is equivalent to:

```

SELECT SL
FC
WC
GROUP BY GROUPING SETS (
    ( GCR1, GCR2, ..., GCRn ),
    ( GCR1, GCR2, ..., GCRn-1 ),
    ( GCR1, GCR2, ..., GCRn-2 ),
    ...
    ( GCR1 ),
    ( ) )
HC

```

NOTE 102 – The resulting <group by clause> is a <grouping set list> that contains a <grouping set> for every proper sublist of <grouping column reference list> of the <rollup list> by dropping elements from the right, one by one.

- 8) If <grouping specification> immediately contains <cube list>, then let GCR_i range over the n <grouping column reference>s contained in the <cube list>. *QS* is equivalent to:

```

SELECT SL
FC
WC
GROUP BY GROUPING SETS (
  ( GCR1, GCR2, ..., GCRn-2, GCRn-1, GCRn ),
  ( GCR1, GCR2, ..., GCRn-2, GCRn-1 ),
  ( GCR1, GCR2, ..., GCRn-2, GCRn ),
  ( GCR1, GCR2, ..., GCRn-2 ),
  ...
  ( GCR1 ),
  ( GCR2, ..., GCRn-2, GCRn-1, GCRn ),
  ( GCR2, ..., GCRn-2, GCRn-1 ),
  ( GCR2, ..., GCRn-2, GCRn ),
  ( GCR2, ..., GCRn-2 ),
  ...
  ( GCR2 ),
  ( GCR3, ..., GCRn-2, GCRn-1, GCRn ),
  ( GCR3, ..., GCRn-2, GCRn-1 ),
  ( GCR3, ..., GCRn-2, GCRn ),
  ( GCR3, ..., GCRn-2 ),
  ...
  ( GCR3 ),
  ...
  ( GCRn ),
  ( ) )
HC

```

NOTE 103 – The resulting <group by clause> is a <grouping set list> that contains a <grouping set> for all possible combinations of the grouping columns in the <grouping column reference list> of the <cube list>.

- 9) If <group by clause> contains a <grouping set list> that contains one or more <rollup list>s or <cube list>s, then:
 - a) Let m be the number of <grouping set>s contained in the <grouping set list>.
 - b) Let GS_i , $1 \leq i \leq m$, range over the <grouping set>s contained in the <grouping set list>.
 - c) For each GS_i :
 - i) If GS_i is a <grand total>, then let ni be 0 (zero). If GS_i is a <grouping column reference>, then let ni be 1 (one). Otherwise, let ni be the number of <grouping column reference>s contained in the <grouping column reference list>.
 - ii) Let $GCR_{i,j}$, $1 \leq j \leq ni$, range over the <grouping column reference>s contained in GS_i in the order in which they occur in GS_i .

NOTE 104 – Column references within GCR_i need not be distinct. That is, it may be the case that $GCR_{i,x} = GCR_{i,y}$ for some $x \neq y$.
 - iii) Case:
 - 1) If GS_i is an <ordinary grouping set> or a <grand total>, then let $GSSUB_i$ be the GS_i .
 - 2) If GS_i is a <rollup list>, then let $GSSUB_i$ be:

```
( GCRi,1, GCRi,2, ..., GCRi,ni ),
( GCRi,1, GCRi,2, ..., GCRi,ni-1 ),
( GCRi,1, GCRi,2, ..., GCRi,ni-2 ),
...
( GCRi,1 ),
( )
```

NOTE 105 – $GSSUB_i$ is a list of <ordinary grouping set>s that contains a <grouping set> for every proper sublist of the <grouping column reference list> of the <rollup list> by dropping elements from the right, one by one.

3) If GS_i is a <cube list>, then let $GSSUB_i$ be:

```
( GCRi,1, GCRi,2, ...,
  GCRi,ni-2, GCRi,ni-1, GCRi,ni ),
( GCRi,1, GCRi,2, ...,
  GCRi,ni-2, GCRi,ni-1 ),
( GCRi,1, GCRi,2, ...,
  GCRi,ni-2, GCRi,ni ),
( GCRi,1, GCRi,2, ...,
  GCRi,ni-2 ),
...
( GCRi,1 ),
( GCRi,2, ..., GCRi,ni-2,
  GCRi,(ni)-1, GCRi,ni ),
( GCRi,2, ..., GCRi,ni-2,
  GCRi,(ni)-1 ),
( GCRi,2, ..., GCRi,ni-2,
  GCRi,ni ),
( GCRi,2, ..., GCRi,ni-2 ),
...
( GCRi,2 ),
( GCRi,3, ..., GCRi,ni-2,
  GCRi,ni-1, GCRi,ni ),
( GCRi,3, ..., GCRi,ni-2,
  GCRi,ni-1 ),
( GCRi,3, ..., GCRi,ni-2,
  GCRi,ni ),
( GCRi,3, ..., GCRi,ni-2 ),
( GCRi,3 ),
...
( GCRi,ni ),
( )
```

NOTE 106 – $GSSUB_i$ is a list of <ordinary grouping set>s that contains a <grouping set> for all possible combinations of the grouping columns in the <grouping column reference list> of the <cube list>.

iv) QS is equivalent to:

```
SELECT SL
FC
WC
GROUP BY GROUPING SETS ( GSSUB1, GSSUB2, ..., GSSUBm )
HC
```

10) If <grouping specification> immediately contains a <concatenated grouping>, then:

a) Let m be the number of <grouping set>s contained in the <concatenated grouping>.

- b) Let GS_i , $1 \leq i \leq m$, range over the <grouping set>s contained in the <concatenated grouping>.
- c) For each GS_i :
- If GS_i is a <grand total>, then let ni be 0 (zero). If GS_i is a <grouping column reference>, then let ni be 1 (one). Otherwise, let ni be the number of <grouping column reference>s contained in the <grouping column reference list>.
 - Let $GCR_{i,j}$, $1 \leq j \leq ni$, range over the <grouping column reference>s contained in GS_i in the order in which they occur in GS_i .
NOTE 107 – Column references within GCR_i need not be distinct. That is, it may be the case that $GCR_{i,x} = GCR_{i,y}$ for some $x \neq y$.
 - Case:
 - If GS_i is an <ordinary grouping set> or a <grand total>, then let $GSFACTOR_i$ be GS_i .
 - If GS_i is a <rollup list>, then let $GSFACTOR_i$ be:

$$\begin{aligned}
 & (\\
 & \quad (GCR_{i,1}, GCR_{i,2}, \dots, GCR_{i,ni}), \\
 & \quad (GCR_{i,1}, GCR_{i,2}, \dots, GCR_{i,ni-1}), \\
 & \quad (GCR_{i,1}, GCR_{i,2}, \dots, GCR_{i,ni-2}), \\
 & \quad \dots \\
 & \quad (GCR_{i,1}), \\
 & \quad () \\
 &)
 \end{aligned}$$

NOTE 108 – $GSFACTOR_i$ is a list of <ordinary grouping set>s that contains a <grouping set> for every proper sublist of <grouping column reference list> of the <rollup list> by dropping elements from the right, one by one.

- If GS_i is a <cube list>, then let $GSFACTOR_i$ be:

```
(
  ( GCRi,1, GCRi,2, ..., GCRi,ni-2,
    GCRi,ni-1, GCRi,n ),
  ( GCRi,1, GCRi,2, ..., GCRi,ni-2,
    GCRi,ni-1 ),
  ( GCRi,1, GCRi,2, ..., GCRi,ni-2,
    GCRi,n ),
  ( GCRi,1, GCRi,2, ..., GCRi,ni-2 ),
  ...
  ( GCRi,1 ),
  ( GCRi,2, ..., GCRi,ni-2, GCRi,ni-1,
    GCRi,n ),
  ( GCRi,2, ..., GCRi,ni-2, GCRi,ni-1 ),
  ( GCRi,2, ..., GCRi,ni-2, GCRi,n ),
  ( GCRi,2, ..., GCRi,ni-2 ),
  ...
  ( GCRi,2 ),
  ( GCRi,3, ..., GCRi,ni-2, GCRi,ni-1,
    GCRi,n ),
  ( GCRi,3, ..., GCRi,ni-2, GCRi,ni-1 ),
  ( GCRi,3, ..., GCRi,ni-2, GCRi,n ),
  ( GCRi,3, ..., GCRi,ni-2 ),
  ( GCRi,3 ),
  ...
  ( GCRi,ni ),
  ( )
)
```

NOTE 109 – $GSFACTOR_i$ is a list of <ordinary grouping set>s that contains a <grouping set> for all possible combinations of the grouping columns in the <grouping column reference list> of the <cube list>.

d) Let $CGPRODUCT_m$ be defined as follows:

i) Let $CGPRODUCT_1$ be $GSFACTOR_1$.

ii) For i ranging from 2 to m ,

Case:

1) If $CGPRODUCT_{i-1}$ is a <grand total>, then let $CGPRODUCT_i$ be $GSFACTOR_i$.

2) If $GSFACTOR_i$ is a <grand total>, then let $CGPRODUCT_i$ be $CGPRODUCT_{i-1}$.

3) If neither $CGPRODUCT_{i-1}$ nor $GSFACTOR_i$ is a <grand total>, then:

NOTE 110 – $CGPRODUCT_{i-1}$ and $GSFACTOR_i$ can be either a <ordinary grouping set> or a list of <ordinary grouping set>s. $GSFACTOR_i$ will be a list of <ordinary grouping set>s only when the original GS_i was either a <rollup list> or a <cube list>.

A) If $CGPRODUCT_{i-1}$ is an <ordinary grouping set>, then let ST_{i-1} be 1 (one); otherwise, let ST_{i-1} be the number of <ordinary grouping set>s contained in the list of <ordinary grouping set>s.

B) Let $GST_{i-1,j}$, $1 \leq j \leq ST_{i-1}$, range over the <ordinary grouping set>s contained in $CGPRODUCT_{i-1}$. Let $N_{i-1,j}$ be the number of <grouping column reference>s contained in the <grouping column reference list> of $GST_{i-1,j}$.

C) Let $GCRT_{i-1,j,k}$, $1 \leq k \leq N_{i-1,j}$, range over the <grouping column reference>s contained in $GST_{i-1,j}$.

NOTE 111 – Column references within $GCRT_{i-1,j}$ need not be distinct. That is, it may be the case that $GCRT_{i-1,j,x} = GCRT_{i-1,j,y}$ for some $x \neq y$.

- D) If $GSFACTOR_i$ is an <ordinary grouping set>, then let S_i be 1 (one); otherwise, let S_i be the number of <ordinary grouping set>s contained in the list of <ordinary grouping set>s.
- E) Let $GS_{i,j}$, $1 \leq j \leq S_i$, range over the <ordinary grouping set>s contained in $GSFACTOR_i$. Let $N_{i,j}$ be the number of <grouping column reference>s contained in the <grouping column reference list> of $GS_{i,j}$.
- F) Let $GCR_{i,j,k}$, $1 \leq k \leq N_{i,j}$, range over the <grouping column reference>s contained in $GS_{i,j}$.

NOTE 112 – Column references within $GCR_{i,j}$ need not be distinct. That is, it may be the case that $GCR_{i,j,x} = GCR_{i,j,y}$ for some $x \neq y$.

- G) $CGPRODUCT_i$ is a list of <ordinary grouping sets> constructed as follows:

```
(
  ( GCRTi-1, 1, 1,
    GCRTi-1, 1, 2, ...
    GCRTi-1, 1, Ni-1,1,
    GCRi, 1, 1,
    GCRi, 1, 2, ...
    GCRi, 1, Ni,1 ),
  ( GCRTi-1, 1, 1,
    GCRTi-1, 1, 2, ...
    GCRTi-1, 1, Ni-1,1,
    GCRi, 2, 1,
    GCRi, 2, 2, ...
    GCRi, 2, Ni,2 ),
  ...
  ( GCRTi-1, 1, 1,
    GCRTi-1, 1, 2, ...
    GCRTi-1, 1, Ni-1,1,
    GCRi, Si, 1,
    GCRi, Si, 2, ...
    GCRi, Si, Ni,Si ),
  ( GCRTi-1, 2, 1,
    GCRTi-1, 2, 2, ...
    GCRTi-1, 2, Ni-1,2,
    GCRi, 1, 1,
    GCRi, 1, 2, ...
    GCRi, 1, Ni,1 ),
  ( GCRTi-1, 2, 1,
    GCRTi-1, 2, 2, ...
    GCRTi-1, 2, Ni-1,2,
    GCRi, 2, 1,
    GCRi, 2, 2, ...
    GCRi, 2, Ni,2 ),
  ...
  ( GCRTi-1, 2, 1,
    GCRTi-1, 2, 2, ...
    GCRTi-1, 2, Ni-1,2,
    GCRi, Si, 1,
    GCRi, Si, 2, ...
    GCRi, Si, Ni,Si ),
```

```

...
( GCRTi-1, STi, 1,
  GCRTi-1, STi, 2, ...
  GCRTi-1, STi, Ni-1, STi'
  GCRi, 1, 1,
  GCRi, 1, 2, ...
  GCRi, 1, Ni,1 ),
( GCRTi-1, STi, 1,
  GCRTi-1, STi, 2, ...
  GCRTi-1, STi, Ni-1, STi'
  GCRi, 2, 1,
  GCRi, 2, 2, ...
  GCRi, 2, Ni,2 ),
...
( GCRTi-1, STi, 1,
  GCRTi-1, STi, 2, ...
  GCRTi-1, STi, Ni-1, STi'
  GCRi, Si, 1,
  GCRi, Si, 2, ...
  GCRi, Si, Ni, Si )
)

```

- e) If $CGPRODUCT_m$ is <grand total> or an <ordinary grouping set>, then let $CGPRODUCT_m$ be ($CGPRODUCT_m$).
- f) QS is equivalent to:

```

SELECT SL
FC
WC
GROUP BY GROUPING SETS  $CGPRODUCT_m$ 
HC

```

- 11) If <grouping specification> immediately contains a <grouping set list> that contains only <ordinary grouping set>s or <grand total>, then:
- Let m be the number of <grouping set>s contained in the <grouping set list>.
 - Let GS_i , $1 \leq i \leq m$, range over the <grouping set>s contained in the <grouping set list>.
 - Let p be the number of distinct <column reference>s that are contained in the <group by clause>.
 - Let PC be an ordered list of these <column reference>s ordered according to their left-to-right occurrence in the list.
 - Let PC_k , $1 \leq k \leq p$, be the k -th <column reference> in PC .
 - Let $DTPC_k$ be the declared type of the column identified by PC_k .
 - Let $CNPC_k$ be the column name of the column identified by PC_k .

h) For each GS_i :

- i) If GS_i is a <grand total>, then let ni be 0 (zero). If GS_i is a <grouping column reference>, then let ni be 1 (one). Otherwise, let ni be the number of <grouping column reference>s contained in the <grouping column reference list>.
- ii) Let $GCR_{i,j}$, $1 \leq j \leq ni$, range over the <grouping column reference>s contained in GS_i .
- iii) Let $COMMON\_TABLE$ be an implementation-dependent <query name> not equivalent to any other <query name> contained in the innermost containing <query expression>.

iv) Case:

1) If GS_i is an <ordinary grouping set>, then

A) Case:

I) If $PC_k = GCR_{i,j}$ for some j , $1 \leq j \leq ni$, then let $PC_{i,k}$ be $CNPC_k$ and let $PCBIT_{i,k}$ be 0 (zero).

II) Otherwise, let $PC_{i,k}$ be

$CAST(NULL AS DTPC_k) AS CNPC_k$

and let $PCBIT_{i,k}$ be 1 (one).

NOTE 113 – $PC_{i,k}$ is the <select sublist> representative of PC_k for GS_i and $PCBIT_{i,k}$ is the grouping function result of PC_k for GS_i .

B) Let $GSSQL_i$ be:

```
SELECT
  PCi,1, PCBITi,1,
  PCi,2, PCBITi,2,
  ...
  PCi,p, PCBITi,p,
  SING
FROM COMMON_TABLE
GROUP BY GCRi,1, ..., GCRi,ni
```

2) If GS_i is a <grand total>, then let $GSSQL_i$ be

```
SELECT CAST(NULL AS DTPC1)
      AS CNPC1, 1,
      CAST(NULL AS DTPC2)
      AS CNPC2, 1,
      .
      .
      .
      CAST(NULL AS DTPCp)
      AS CNPCp, 1,
      SING
FROM COMMON_TABLE
```

7.9 <group by clause>

v) QS is equivalent to:

```

WITH
  COMMON_TABLE AS
    ( SELECT * FC WC ),
  GROUP_BY_RESULT1 AS
    ( GSSQL1
      UNION ALL
      GSSQL2
      UNION ALL
      .
      .
      .
      UNION ALL
      GSSQLm )
SELECT SL FROM GROUP_BY_RESULT1 HC

```

NOTE 114 – GROUPBYRESULT₁ is the group-by result of QS.

- 12) If the <grouping specification> specified by <group by clause> GBC contains no <rollup list>, <cube list>, or <grouping sets list>, then let GCR_i range over the n <grouping column reference>s contained in the <group by clause>. Let GBR_0 be:

```

SELECT GCR1, 0, ..., GCRn, 0, SING
FC
WC
GBC

```

GBR_0 is the group-by result of QS.

Access Rules

None.

General Rules

- 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 for two rows of T results in unknown, 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 to a grouping column is a reference to the common value in that column of the rows in that group.

NOTE 115 – Where application of the General Rules of Subclause 8.2, "<comparison predicate>", results in the formation of a group with values that are equal, but of different lengths or containing

different sequences of characters in the same grouping column, the value selected as the common value of that grouping column in that group is implementation-dependent. See Subclause 8.2, "<comparison predicate>".

Conformance Rules

- 1) Without Feature T431, "CUBE and ROLLUP", conforming SQL language shall not specify ROLLUP or CUBE.
- 2) Without Feature F691, "Collation and translation", conforming SQL language shall not contain any <collate clause>.
- 3) Without Feature S024, "Enhanced structured types", a <column reference> simply contained in a <group by clause> shall not reference a column of a structured type.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

7.10 <having clause>

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 unambiguously reference a column that is functionally dependent on *G* or be an outer reference.
NOTE 116 – *Outer reference* is defined in Subclause 6.6, “<column reference>”.
- 4) Each column reference contained in a <subquery> in the <search condition> that references a column of *T* shall reference a column that is functionally dependent on *G* or shall be specified within a <set function specification>.
- 5) The <having clause> is *possibly non-deterministic* if it contains a reference to a column *C* of *T* that has a data type of datetime with a time zone displacement value, character string, or user-defined type and at least one of the following is true:
 - a) *C* is specified within a <set function specification> that specifies MIN or MAX.
 - b) *C* is functionally dependent on *G*.
- 6) The row type of the result of the <having clause> is the row type *RT* of *T*.

Access Rules

None.

General Rules

- 1) The <search condition> is applied to each group of *R*. The result of the <having clause> is a grouped table of those groups of *R* for which the result of the <search condition> is true.
- 2) When the <search condition> is applied to a given group of *R*, that group is the argument or argument source of each <set function specification> directly contained in the <search condition>, unless the <column reference> in the <set function specification> is an outer reference.

- 3) Each <subquery> in the <search condition> is effectively evaluated for each group of *R* and the result used in the application of the <search condition> to the given group of *R*. If any evaluated <subquery> contains an outer reference to a column of *T*, then the reference is to the values of that column in the given group of *R*.

NOTE 117 – *Outer reference* is defined in Subclause 6.6, “<column reference>”.

Conformance Rules

None.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

7.11 <query specification>

Function

Specify a table derived from the result of a <table expression>.

Format

```
<query specification> ::=  
    SELECT [ <set quantifier> ] <select list>  
    <table expression>  
  
<select list> ::=  
    <asterisk>  
    | <select sublist> [ { <comma> <select sublist> }... ]  
  
<select sublist> ::=  
    <derived column>  
    | <qualified asterisk>  
  
<qualified asterisk> ::=  
    <asterisked identifier chain> <period> <asterisk>  
    | <all fields reference>  
  
<asterisked identifier chain> ::=  
    <asterisked identifier> [ { <period> <asterisked identifier> }... ]  
  
<asterisked identifier> ::= <identifier>  
  
<derived column> ::=  
    <value expression> [ <as clause> ]  
  
<as clause> ::= [ AS ] <column name>  
  
<all fields reference> ::=  
    <value expression primary> <period> <asterisk>
```

Syntax Rules

- 1) Let T be the result of the <table expression>.
- 2) Let TQS be the table that is the result of a <query specification>.
- 3) Case:
 - a) If the <select list> “*” is simply contained in a <subquery> that is immediately contained in an <exists predicate>, then the <select list> is equivalent to a <value expression> that is an arbitrary <literal>.
 - b) Otherwise, the <select list> “*” is equivalent to a <value expression> sequence in which each <value expression> is a column reference that references a column of T and each column of T is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within T .
- 4) The degree of the table specified by a <query specification> is equal to the cardinality of the <select list>.

- 5) If a <set quantifier> DISTINCT is specified and one of the columns of T has a declared type DT that is a user-defined type, then the comparison form of DT shall be FULL.
- 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 , $1 \text{ (one)} \leq i \leq N$, 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 within the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter SP whose <identifier> 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 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 .
 - iii) 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 .
 - iv) 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 .
 - v) 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:

$$(PIC_{BL}) . I_{BL+1} . \dots . I_N . *$$

NOTE 118 – 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 $F_1, \dots, F_n$ be the field names of VER . AFR is equivalent to:

$$VEP . F_1 , \dots , VEP . F_n$$

- 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 directly contained in each <value expression> and each column reference contained in a <set function specification> directly contained in each <value expression> shall unambiguously reference a column of T .
- 11) 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> contains a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.
 - c) The <query specification> directly contains a <having clause> that is possibly non-deterministic.
 - d) The <select list> 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 either
 - i) C is specified with a <set function specification> that specifies MIN or MAX, or
 - ii) C is a grouping column of T .
- 12) If <table expression> does not immediately contain a <group by clause> and <select list> contains either a <value expression> that contains a <set function specification> that contains a reference to a column of T or a <value expression> that directly contains a <set function specification> that does not contain an outer reference, then GROUP BY () is implicit.

- 13) If T is a grouped table, then let G be the set consisting of every column referenced by a <column reference> contained in the <group by clause> immediately contained in <table expression>. In each <value expression>, 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 a <set function specification>.
- 14) 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> from which the column was derived.
- 15) 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 and not equivalent to the <column name> of any column, other than itself, of a table referenced by any <table reference> contained in the SQL-statement.
- 16) A column of TQS is *known not null* if and only if one of the following conditions apply:
- a) It does not contain any of the following:
 - i) A column reference for a column C that is possibly nullable.
 - ii) An <indicator parameter>.
 - iii) An SQL parameter.
 - iv) 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.
 - v) A <subquery>.
 - vi) CAST (NULL AS X) (where X represents a <data type> or a <domain name>).
 - vii) CURRENT_USER, CURRENT_ROLE, or SYSTEM_USER.
 - viii) A <set function specification> that does not contain COUNT.
 - ix) A <case expression>.
 - x) A <field reference>.
 - xi) An <element reference>.
 - xii) A <dereference operation>.
 - xiii) A <reference resolution>.

- b) An implementation-defined rule by which the SQL-implementation can correctly deduce that the value of the column cannot be null.
- 17) Let *TREF* be the <table reference>s that are simply contained in the <from clause> of the <table expression>. The *simply underlying tables* of the <query specification> are the <table or query name>s and <derived table>s contained in *TREF* without an intervening <derived table>.
- 18) The terms *key-preserving* and *one-to-one* are defined as follows:
- a) Let *UT* denote some simply underlying table of *QS*, let *UTCOLS* be the set of columns of *UT*, let *QSCOLS* be the set of columns of *QS*, and let *QSCN* be an exposed <table name> or exposed <correlation name> for *UT* whose scope clause is *QS*.
NOTE 119 – “strong candidate key” is defined in Subclause 4.19, “Candidate keys”.
- b) *QS* is said to be *key-preserving with respect to UT* if there is some strong candidate key *CKUT* of *UT* such that every member of *CKUT* has some counterpart under *QSCN* in *QSCOLS*.
NOTE 120 – “Counterpart” is defined in Subclause 4.18.1, “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 121 – 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*.
- 19) A <query specification> is *potentially updatable* if and only if the following conditions hold:
- a) DISTINCT is not specified.
- b) Of those <derived column>s in the <select list> that are column references, no column reference appears more than once in the <select list>.
- c) The <table expression> immediately contained in *QS* does not simply contain a <group by clause> or a <having clause>.
- 20) A <query specification> *QS* is *insertable-into* if and only if every simply underlying table of *QS* is *insertable-into*.
- 21) If a <query specification> *QS* is *potentially updatable*, then
Case:
- a) If the <from clause> of the <table expression> specifies exactly one <table reference>, then a column of *QS* is said to be an *updatable column* if it has a counterpart in *TR* that is updatable.
NOTE 122 – The notion of updatable columns of table references is defined in Subclause 7.6, “<table reference>”.
- b) Otherwise, a column of *QS* is said to be an *updatable column* if it has a counterpart in some column of some simply underlying table *UT* of *QS* such that *QS* is one-to-one with respect to *UT*.

- 22) A <query specification> is *updatable* if it is potentially updatable and it has at least one updatable column.
- 23) A <query specification> *QS* is *simply updatable* if it is updatable, the <from clause> immediately contained in the <table expression> immediately contained in *QS* contains exactly one <table reference>, and every result column of *QS* is updatable.
- 24) 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) 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>.

Case:

- i) If the <set quantifier> **DISTINCT** is not specified, then the result of the <query specification> is *TEMP*.
- ii) 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.

b) If *T* is a grouped table, then

Case:

- i) If *T* has 0 (zero) groups, then the result of the <query specification> is 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 or argument source of each <set function specification> in the <value expression>.

Case:

- 1) If the <set quantifier> **DISTINCT** is not specified, then the result of the <query specification> is *TEMP*.
- 2) 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.

Conformance Rules

- 1) Without Feature F801, “Full set function”, the <set quantifier> DISTINCT shall not be specified more than once in a <query specification>, excluding any <subquery> of that <query specification>.
- 2) Without Feature T051, “Row types”, conforming SQL language shall not specify <all fields reference>.
- 3) Without Feature T301, “Functional dependencies”, if *T* is a grouped table, then in each <value expression>, each <column reference> that references a column of *T* shall reference a grouping column or be specified in a <set function specification>.
- 4) Without Feature S024, “Enhanced structured types”, if any column in the result of a <query specification> is of structured type, then DISTINCT shall not be specified or implied.
- 5) Without Feature T111, “Updatable joins, unions, and columns”, a <query specification> QS is not updatable if it is not simply updatable.

7.12 <query expression>

Function

Specify a table.

Format

```

<query expression> ::=
    [ <with clause> ] <query expression body>

<with clause> ::= WITH [ RECURSIVE ] <with list>

<with list> ::=
    <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
    <query name>
    [ <left paren> <with column list> <right paren> ]
    AS <left paren> <query expression> <right paren>
    [ <search or cycle clause> ]

<with column list> ::= <column name list>

<query expression body> ::=
    <non-join query expression>
    | <joined table>

<non-join query expression> ::=
    <non-join query term>
    | <query expression body> UNION [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>
    | <query expression body> EXCEPT [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>

<query term> ::=
    <non-join query term>
    | <joined table>

<non-join query term> ::=
    <non-join query primary>
    | <query term> INTERSECT [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query primary>

<query primary> ::=
    <non-join query primary>
    | <joined table>

<non-join query primary> ::=
    <simple table>
    | <left paren> <non-join query expression> <right paren>

<simple table> ::=
    <query specification>
    | <table value constructor>
    | <explicit table>

```

7.12 <query expression>

<explicit table> ::= TABLE <table name>

<corresponding spec> ::=
CORRESPONDING [BY <left paren> <corresponding column list> <right paren>]

<corresponding column list> ::= <column name list>

Syntax Rules

1) If <with clause> is specified, then:

- a) If a <with clause> *WC* immediately contains **RECURSIVE**, then *WC* 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 and let WLE_i and WLE_j be the *i*-th and *j*-th <with list element>s for every (*i,j*) with *i* ranging from 1 (one) to *n* and *j* ranging from *i*+1 to *n*. If WLE_i is not potentially recursive, then it shall not immediately contain the <query name> immediately contained in WLE_j .
- 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 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 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) No column in *WQT* shall have a coercibility characteristic of *No collating sequence*.
- 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 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 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 $WQN_i, \dots, WQN_k$, with $k \neq i$, then it is said that $WQN_i, \dots, WQN_k$ are *recursive* and *mutually recursive* to each other, $WQT_i, \dots, WQT_k$ are *recursive* and *mutually recursive* to each other, and $WLE_i, \dots, WLE_k$ are *recursive* and *mutually recursive* to each other.

For each j ranging from i to k , WQN_j belongs to the stratum of $WQE_i, \dots$, and WQE_k .

- 3) Among the $WQE_i, \dots, WQE_k$ of a given stratum, there shall be at least one <query expression>, say WQE_j , such that:

- A) WQE_j is a <non-join query expression> that immediately contains UNION.
- B) WQE_j has one operand that does not contain $WQN_i, \dots, WQN_k$. 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 $WQN_i, \dots, WQN_k$. For any anchor name WQN_j , remove the arcs to those query names WQN_l that are contained in WQE_j . The remaining graph $SCCGP$ shall not contain any cycle.

ii) If WLE_i is recursive, then:

- 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 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 both WQN_k and WQN_l .
- II) WQE_i does not contain a <joined table> such that $TR1$ and $TR2$ are the first and second <table reference>s, respectively, and $TR1$ and $TR2$ contain WQN_k and WQN_l , respectively, except for union join.
- III) WQE_i does not contain a <table expression> that immediately contains a <from clause> that contains WQN_k , and immediately contains a <where clause> containing a <subquery> that contains 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 <non-join query expression> that contains 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 WQN_j .
 - 3) WQE_i shall not contain a <table subquery> TSQ that contains WQN_j , unless TSQ is immediately contained in a <table reference> that is immediately contained in a <table expression> that is immediately contained in a <query specification> that is immediately contained in a <non-join query expression> 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 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 <non-join query term> that contains WQN_j and immediately 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> that contains WQN_j .
 - B) QJ immediately contains a <join type> that specifies LEFT and a <table reference> following the <join type> that contains WQN_j .
 - C) QJ immediately contains a <join type> that specifies RIGHT and a <table reference> preceding the <join type> that contains 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 WQN_j .
 - B) QJ immediately contains a <join type> that specifies LEFT and a <table primary> following the <join type> that contains WQN_j .
 - C) QJ immediately contains a <join type> that specifies RIGHT and a <table reference> preceding the <join type> that contains 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 <non-join query expression> that immediately contains UNION or UNION ALL. Let QEL_i and QTR_i be the <query expression> and the <query term> immediately contained in WQE_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>.
- 2) Let T be the table specified by the <query expression>.
- 3) The <explicit table>
- TABLE <table name>
- is equivalent to the <query expression>
- (SELECT * FROM <table name>)
- 4) Let *set operator* be UNION ALL, UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT.
- 5) If UNION, EXCEPT, or INTERSECT is specified and neither ALL nor DISTINCT is specified, then DISTINCT is implicit.
- 6) <query expression> $QE1$ is *updatable* if and only if for every <query expression> or <query specification> $QE2$ that is simply contained in $QE1$:
- a) $QE1$ contains $QE2$ without an intervening <non-join query expression> that specifies UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.
 - b) If $QE1$ simply contains a <non-join query expression> $NJQE$ that specifies UNION ALL, then:
 - i) $NJQE$ immediately contains a <query expression> LO and a <query term> RO such that no leaf generally underlying table of LO is also a leaf generally underlying table of RO .
 - ii) For every column of $NJQE$, the underlying columns in the tables identified by LO and RO , respectively, are either both updatable or not updatable.
 - c) $QE1$ contains $QE2$ without an intervening <non-join query term> that specifies INTERSECT.
 - d) $QE2$ is updatable.
- 7) <query expression> $QE1$ is *insertable-into* if and only if $QE1$ simply contains exactly one <query expression> or <query specification> $QE2$ and $QE2$ is insertable-into.
- 8) Case:
- a) If a <simple table> is a <query specification>, then the column descriptor of the i -th column of the <simple table> is the same as the column descriptor of the i -th column of the <query specification>.
 - b) If a <simple table> is an <explicit table>, then the column descriptor of the i -th column of the <simple table> is the same as the column descriptor of the i -th column of the table identified by the <table name> contained in the <explicit table>.

7.12 <query expression>

- c) Otherwise, the column descriptor of the i -th column of the <simple table> is the same as the column descriptor of the i -th column of the <table value constructor>, except that the <column name> is implementation-dependent and not equivalent to the <column name> of any column, other than itself, of any table referenced by a <table reference> contained in the outermost SQL-statement.
- 9) Case:
- a) If a <non-join query primary> is a <simple table>, then the column descriptor of the i -th column of the <non-join query primary> is the same as the column descriptor of the i -th column of that <simple table>.
- b) Otherwise, the column descriptor of the i -th column of the <non-join query primary> is the same as the column descriptor of the i -th column of the <non-join query expression>.
- 10) Case:
- a) If a <query primary> is a <non-join query primary>, then the column descriptor of the i -th column of the <query primary> is the same as the column descriptor of the i -th column of that <non-join query primary>.
- b) Otherwise, the column descriptor of the i -th column of the <query primary> is the same as the column descriptor of the i -th column of the <joined table>.
- 11) If a set operator is specified in a <non-join query term> or a <non-join query expression>, then:
- a) Let $T1$, $T2$, and TR be respectively the first operand, the second operand, and the result of the <non-join query term> or <non-join query expression>.
- b) Let $TN1$ and $TN2$ be the effective names for $T1$ and $T2$, respectively.
- c) If the set operator is UNION DISTINCT, EXCEPT DISTINCT, or INTERSECT DISTINCT and any column of $T1$ or $T2$ has a declared type DT that is a user-defined type, then the comparison form of DT shall be FULL.
- 12) If a set operator is specified in a <non-join query term> or a <non-join query expression>, 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 <non-join query term> or <non-join query expression> is equivalent to:
- $$(\text{ SELECT } SL \text{ FROM } TN1) \text{ OP } (\text{ SELECT } SL \text{ FROM } TN2)$$
- b) If CORRESPONDING is not specified, then *T1* and *T2* shall be of the same degree.
- 13) If the <non-join query term> is a <non-join query primary>, then the declared type of the <non-join query term> is that of the <non-join query primary>. If the <non-join query primary> has column descriptors, then the column descriptor of the *i*-th column of the <non-join query term> is the same as the column descriptor of the *i*-th column of the <non-join query primary>.
- 14) If the <non-join query term> immediately contains a set operator, then
- a) Case:
- 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*.
- ii) Otherwise, the <column name> of the *i*-th column of *TR* is implementation-dependent and not equivalent to the <column name> of any column, other than itself, of any table referenced by any <table reference> contained in the SQL-statement.
- b) The declared type of the *i*-th column of *TR* is determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of the *i*-th column of *T1* and the *i*-th column of *T2*. If the *i*-th columns of either *T1* or *T2* are known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.
- 15) Case:
- a) If a <query term> is a <non-join query term> that has column descriptors, then 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 <non-join query term>.
- b) If the <query term> is a <joined table>, then 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 <joined table>.
- 16) Case:
- a) If the <non-join query term> has column descriptors, then the column descriptor of the *i*-th column of the <non-join query expression> is the same as the column descriptor of the *i*-th column of the <non-join query term>.
- b) If a <non-join query expression> immediately contains a set operator, then
- i) Case:
- 1) 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*.

7.12 <query expression>

- 2) Otherwise, the <column name> of the i -th column of TR is implementation-dependent and not equivalent to the <column name> of any column, other than itself, of any table referenced by any <table reference> contained in the SQL-statement.
- ii) If TR is not the result of an anchor expression, then the declared type of the i -th column of TR is determined by applying the Syntax Rules of Subclause 9.3, "Data types of results of aggregations", to the declared types of the i -th column of $T1$ and the i -th column of $T2$.

Case:

- 1) If the <non-join query expression> 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 , $1 \leq k \leq l$, such that a column of WQT_k is an underlying column of the i -th column of $TREC$. Otherwise, that column is said to be *not recursively referred to*.
 - 2) If the i -th column of TR is not recursively referred to, then the declared type of the i -th column of TR is determined by applying Subclause 9.3, "Data types of results of aggregations", to the declared types of the i -th column of $T1$ and the i -th column of $T2$. If the i -th columns of either $T1$ or $T2$ are known not nullable, then the i -th column of TR is *known not nullable*; otherwise, the i -th column of TR is *possibly nullable*.
 - 3) If the i -th column of TR is recursively referred to, then:
 - A) The i -th column of TR is *possibly nullable*.
 - B) Case:
 - I) If $T1$ is $TNREC$, then if the i -th column of TR is recursively referred to, then the declared type of the i -th column of TR is the same as the declared type of the i -th column of $T1$.
 - II) If $T2$ is $TNREC$, then if the i -th column of TR is recursively referred to, then the declared type of the i -th column of TR is the same as the declared type of the i -th column of $T2$.

17) Case:

- a) If a <query expression body> is a <non-join query expression> that has column descriptors, then the column descriptors of the i -th column of the <query expression body> and of the

immediately containing <query expression> are the same as the column descriptor of the *i*-th column of the <non-join query expression>.

- b) If a <query expression body> is a <joined table>, then the column descriptors of the *i*-th column of the <query expression body> and of the immediately containing <query expression> are the same as the column descriptor of the *i*-th column of the <joined table>.
- 18) The *simply underlying tables* 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*.
- 19) A <query expression> is *possibly non-deterministic* if any of the following are true:
- a) The <query expression> is a <non-join query primary> that is possibly non-deterministic.
 - b) The <query expression> is a <joined table> that is possibly non-deterministic.
 - c) UNION, EXCEPT, or INTERSECT is specified and either of the first or second operands is possibly non-deterministic.
 - 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) Either of the following are 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 a time zone displacement.
- 20) The *underlying columns* of each column of *QE* and of *QE* itself are defined as follows:
- a) A column of a <table value constructor> has no underlying columns.
 - b) The underlying columns of every *i*-th column of a <simple table> *ST* are the underlying columns of the *i*-th column of the table immediately contained in *ST*. A column of *ST* is called an *updatable column* of *ST* if the underlying column of *ST* is updatable; otherwise, this column is *not updatable*.
 - c) If no set operator is specified, then the underlying columns of every *i*-th column of *QE* are the underlying columns of the *i*-th column of the <simple table> simply contained in *QE*. A column of such a *QE* is called an *updatable column* of *QE* if its underlying column is updatable; otherwise, this column is *not updatable*.
 - d) If a set operator is specified, then the underlying columns of every *i*-th column of *QE* are the underlying columns of the *i*-th column of *T1* and those of the *i*-th column of *T2*. If a set operator UNION ALL is specified, then a column of that *QE* is called an *updatable column* of *QE* if both its underlying columns of *T1* and *T2* are updatable.
 - 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*.
- 21) If the declared type of any column of a <query term> is large object string, then ALL shall be specified.

7.12 <query expression>

- 22) If the declared type of any column of a <query primary> is large object string, then ALL shall be specified.
- 23) A <query expression> *QE* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data, unless *QE* is a <table value constructor> and is immediately contained in an <insert columns and source> that is immediately contained in an <insert statement>.

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*, and let *WQT_i* be the table resulting from the evaluation of *WQE_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 the <query name> *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_k*, $1 \leq k \leq p$ 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 <non-join query expression> 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 <non-join query expression> or <non-join 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 123 – 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

7.12 <query expression>

- 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> or <joined table>.
- b) If a set operator is specified, then the result of applying the set operator is a table containing the following rows:

- i) Let R be a row that is a duplicate of some row in $T1$ or of some row in $T2$ or both. Let m be the number of duplicates of R in $T1$ and let n be the number of duplicates of R in $T2$, where $m \geq 0$ and $n \geq 0$.

- ii) If DISTINCT is specified or implicit, then

Case:

- 1) If UNION is specified, then

Case:

A) If $m > 0$ or $n > 0$, then T contains exactly one duplicate of R .

B) Otherwise, T contains no duplicate of R .

- 2) If EXCEPT is specified, then

Case:

A) If $m > 0$ and $n = 0$, then T contains exactly one duplicate of R .

B) Otherwise, T contains no duplicate of R .

- 3) If INTERSECT is specified, then

Case:

A) If $m > 0$ and $n > 0$, then T contains exactly one duplicate of R .

B) Otherwise, T contains no duplicates of R .

- iii) If ALL is specified, then

Case:

- 1) If UNION is specified, then the number of duplicates of R that T contains is $(m + n)$.

- 2) If EXCEPT is specified, then the number of duplicates of R that T contains is the maximum of $(m - n)$ and 0 (zero).

- 3) If INTERSECT is specified, then the number of duplicates of R that T contains is the minimum of m and n .

NOTE 124 – See the General Rules of Subclause 8.2, "<comparison predicate>".

- 4) If a set operator is specified, then for each column whose declared type is interval, let *UDT* be in turn the declared type of the corresponding column of *T* and let *SV* be the value of the column in each row of the first and second operands. The value of the corresponding column of *T* in the corresponding row of *T* is

CAST (*SV* AS *UDT*)

- 5) Case:

- a) If EXCEPT is specified and a row *R* of *T* is replaced by some row *RR*, then the row of *T1* from which *R* is derived is replaced by *RR*.
- b) If INTERSECT is specified, then:
 - i) If a row *R* is inserted into *T*, then:
 - 1) If *T1* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T1*.
 - 2) If *T1* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T1*.
 - 3) If *T2* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T2*.
 - 4) If *T2* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T2*.
 - ii) If a row *R* is replaced by some row *RR*, then:
 - 1) The row of *T1* from which *R* is derived is replaced with *RR*.
 - 2) The row of *T2* from which *R* is derived is replaced with *RR*.

Conformance Rules

- 1) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", a <query expression> shall not specify a <with clause>.
- 2) Without Feature T131, "Recursive query", a <query expression> shall not specify RECURSIVE.
- 3) Without Feature F661, "Simple tables", a <simple table> shall not be a <table value constructor> except in an <insert statement>.
- 4) Without Feature F661, "Simple tables", conforming SQL language shall contain no <explicit table>.
- 5) Without Feature F302, "INTERSECT table operator", a <query term> shall not specify INTERSECT.
- 6) Without Feature F301, "CORRESPONDING in query expressions", a <query expression> shall not specify CORRESPONDING.
- 7) Without Feature T551, "Optional key words for default syntax", conforming SQL language shall contain no explicit UNION DISTINCT, EXCEPT DISTINCT, or INTERSECT DISTINCT.

7.12 <query expression>

- 8) Without Feature S024, “Enhanced structured types”, if any column in the result of a <query expression> is of structured type, then DISTINCT shall not be specified or implied, and neither INTERSECT nor EXCEPT shall be specified.
- 9) Without Feature T111, “Updatable joins, unions, and columns”, a <non-join query expression> that immediately contains UNION is not updatable.
- 10) Without Feature F304, “EXCEPT ALL table operator”, a <query expression> shall not specify EXCEPT ALL.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

7.13 <search or cycle clause>

Function

Specify the generation of ordering and cycle detection information in the result of recursive query expressions.

Format

```

<search or cycle clause> ::=
    <search clause>
  | <cycle clause>
  | <search clause> <cycle clause>

<search clause> ::=
    SEARCH <recursive search order> SET <sequence column>

<recursive search order> ::=
    DEPTH FIRST BY <sort specification list>
  | BREADTH FIRST BY <sort specification list>

<sequence column> ::= <column name>

<cycle clause> ::=
    CYCLE <cycle column list>
    SET <cycle mark column> TO <cycle mark value>
    DEFAULT <non-cycle mark value>
    USING <path column>

<cycle column list> ::=
    <cycle column> [ { <comma> <cycle column> }... ]

<cycle column> ::= <column name>

<cycle mark column> ::= <column name>

<path column> ::= <column name>

<cycle mark value> ::= <value expression>

<non-cycle mark value> ::= <value expression>

```

Syntax Rules

- 1) Let *WLEC* be an expandable <with list element> immediately containing a <search or cycle clause>.
- 2) Let *WQN* be the <query name>, *WCL* the <with column list>, and *WQE* the <query expression> immediately contained in *WLEC*. Let *OP* be the set operator immediately contained in *WQE*, and let *TLO* and *TRO* be the first and the second operand of *OP*, respectively.
 - a) Let *TROSL* be the <select list> of *TRO*. Let *WQNTR* be the <table reference> immediately 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* immediately contains a <correlation name>, then let *WQNCRN* be this correlation name.
- ii) Otherwise, let *WQNCRN* be *WQN*.

b) Case:

- i) If *WLEC* contains a <search clause> *SC*, then let *SQC* be the <sequence column> and *SO* be the <recursive search order> immediately contained in *SC*. Let *SPL* be the <sort specification list> immediately contained in *SO*.

- 1) *SQC* shall not be contained in *WCL*.
- 2) Every <column name> of *SPL* shall also be contained in *WCL*. No <column name> shall be specified more than once in *SPL*.

3) Case:

- A) If *SO* immediately contains *DEPTH*, then let *SCEX1* be

WQNCRN.SQC

let *SCEX2* be

SQC || ARRAY [ROW(*SPL*)]

and let *SCIN* be

ARRAY [ROW(*SPL*)]

- B) If *SO* immediately contains *BREADTH*, then let *SCEX1* be

(SELECT OC.*
FROM (VALUES (*WQNCRN.SQC*))
OC(LEVEL, *SPL*))

let *SCEX2* be

ROW(*SQC*.LEVEL + 1, *SPL*)

and let *SCIN* be

ROW(0, *SPL*)

- ii) If *WLEC* 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 contained in *WCL*. No <column name> shall be specified 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* contains only a <search 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* contains only a <cycle 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* 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) TLOCN(WCL)
  OP
  SELECT ETROSL
  FROM (SELECT ETROSL1 TROTE) TROCRN(EWCL)
  WHERE NCCON
)
```

Access Rules

None.

General Rules

None.

Conformance Rules

None.

7.14 <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 types of the columns of a <row subquery> or <table subquery> are the declared types of the respective columns of *QE*.

Access Rules

None.

General Rules

- 1) If the cardinality of a <row subquery> is greater than 1 (one), then an exception condition is raised: *cardinality violation*.
- 2) 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>.

7.14 <subquery>

- 3) During the evaluation of a <subquery>, an *atomic execution context* is active. When the <subquery> completes, all savepoints that were established during its evaluation are destroyed.

Conformance Rules

- 1) Without Feature T501, “Enhanced EXISTS predicate”, if a <table subquery> is simply contained in an <exists predicate>, then the <select list> of every <query specification> directly contained in the <table subquery> shall comprise either an <asterisk> or a single <derived column>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

8 Predicates

8.1 <predicate>

Function

Specify a condition that can be evaluated to give a boolean value.

Format

```
<predicate> ::=
    <comparison predicate>
    | <between predicate>
    | <in predicate>
    | <like predicate>
    | <null predicate>
    | <quantified comparison predicate>
    | <exists predicate>
    | <unique predicate>
    | <match predicate>
    | <overlaps predicate>
    | <similar predicate>
    | <distinct 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>, <null predicate>, <quantified comparison predicate>, <exists predicate>, <unique predicate>, <match predicate>, <overlaps predicate>, <similar predicate>, <distinct predicate>, or <type predicate>.

Conformance Rules

- 1) Without Feature T141, "SIMILAR predicate", conforming SQL language shall contain no <similar predicate>.
- 2) Without Feature T151, "DISTINCT predicate", conforming SQL language shall contain no <distinct predicate>.
- 3) Without Feature S151, "Type predicate", conforming SQL language shall contain no <type predicate>.

8.1 <predicate>

- 4) Without Feature F741, “Referential MATCH types”, conforming SQL language shall not contain a <match predicate>.
- 5) Without Feature F052, “Intervals and datetime arithmetic”, conforming SQL language shall not contain any <overlaps predicate>.
- 6) Without Feature F291, “UNIQUE predicate”, conforming SQL language shall not contain any <unique predicate>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

8.2 <comparison predicate>

Function

Specify a comparison of two row values.

Format

<comparison predicate> ::=
 <row value expression> <comp op> <row value expression>

<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 expression>s shall be of the same degree.
- 2) Let *corresponding fields* be fields with the same ordinal position in the two <row value expression>s.
- 3) The declared types of the corresponding fields of the two <row value expression>s shall be comparable.
- 4) Let R_x and R_y respectively denote the first and second <row value expression>s.
- 5) Let N be the number of fields in the declared type of R_x . Let X_i , $1 \leq i \leq N$, be the i -th field in the declared type of R_x and let Y_i be the i -th field in the declared type of R_y . For each i :
 - a) If the declared type of X_i or Y_i is large object string, reference type, or array type, then <comp op> shall be either <equals operator> or <not equals operator>.
 - b) Case:
 - i) If the declared types of X_i and Y_i are user-defined types, then:
 - 1) Let $UDT1$ and $UDT2$ be respectively the declared types of X_i and Y_i . $UDT1$ and $UDT2$ shall be in the same type family. $UDT1$ and $UDT2$ shall have comparison types.

NOTE 125 – “Comparison type” is defined in Subclause 4.8.4, “User-defined type comparison and assignment”.

NOTE 126 – The comparison form and comparison categories included in the user-defined type descriptors of both $UDT1$ and $UDT2$ are constrained to be the same — they must be the same throughout a type family. If the comparison category is either STATE or RELATIVE, then the comparison functions of $UDT1$ and $UDT2$ are constrained to be equivalent; if the comparison category is MAP, they are not constrained to be equivalent.

- 2) 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.
- 3) If <less than operator>, <greater than operator>, <less than or equals operator>, or <greater than or equals operator> is specified, then the comparison form of $UDT1$ shall be FULL.

NOTE 127 – If the comparison form is FULL, then the comparison category is constrained to be RELATIVE or MAP; if the comparison form is EQUALS, then the comparison category is also permitted to be STATE.

- ii) If the declared types of X_i and Y_i are character strings, then the pair-wise comparison collating sequence used to compare X_i and Y_i is determined by Subclause 4.2.3, “Rules determining collating sequence usage”. The applicable column in the table shall not indicate “Not permitted: invalid syntax”.
- iii) If the declared types of X_i and Y_i are array 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:

$\text{NOT}(R_x = R_y)$

- b) If the <comp op> is <greater than operator>, then CP is equivalent to:

$(R_y < R_x)$

- c) If the <comp op> is <less than or equals operator>, then CP is equivalent to:

$(R_x < R_y$
OR
 $R_y = R_x)$

- d) If the <comp op> is <greater than or equals operator>, then CP is equivalent to:

$(R_y < R_x$
OR
 $R_y = R_x)$

Access Rules

None.

General Rules

- 1) Let XV and YV be two values represented by <value expression>s X and Y , respectively. The result of:

$$X \text{ <comp op> } Y$$

is determined as follows:

Case:

- a) If either XV or YV is the null value, then

$$X \text{ <comp op> } Y$$

is unknown .

- b) Otherwise,

Case:

- i) If the declared types of XV and YV are row types with degree N , then let X_i , $1 \text{ (one)} \leq i \leq N$, denote a <value expression> whose value and declared type is that of the i -th field of XV and let Y_i denote a <value expression> whose value and declared type is that of the i -th field of YV . The result of

$$X \text{ <comp op> } Y$$

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) $X \text{ <comp op> } Y$ is unknown if $X \text{ <comp op> } Y$ is neither true nor false .

- ii) If the declared types of XV and YV are array types with cardinalities $N1$ and $N2$, respectively, then let X_i , $1 \text{ (one)} \leq i \leq N1$, denote a <value expression> whose value and declared type is that of the i -th element of XV and let Y_i denote a <value expression> whose value and declared type is that of the i -th element of YV . The result of

$$X \text{ <comp op> } Y$$

is determined as follows:

- 1) $X = Y$ is true if $N1 = 0$ (zero) and $N2 = 0$ (zero).

- 2) $X = Y$ is true if $N1 = N2$ and, for all i , $X_i = Y_i$ is true .
- 3) $X = Y$ is false if and only if $N1 \neq N2$ or NOT ($X_i = Y_i$) is true , for some i .
- 4) $X <comp\ op> Y$ is unknown if $X <comp\ op> Y$ is neither true nor false .
- iii) If the declared types of XV and YV are 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> of the comparison function of UDT_x and let $HF2$ be the <routine name> of the comparison function of UDT_y . If $HF1$ is an SQL-invoked function that is a method, then let HFX be $X.HF1$; otherwise, let HFX be $HF1(X)$. If $HF2$ is an SQL-invoked function that is a 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> 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) $X \leq Y$

has the same result as

$$RF(X, Y) = -1 \text{ OR } RF(X, Y) = 0$$

G) $X \geq Y$

has the same result as

$$RF(X, Y) = 1 \text{ OR } RF(X, Y) = 0$$

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

$$SF(X, Y) = \text{TRUE}$$

C) $X \neq Y$

has the same result as

$$SF(X, Y) = \text{FALSE}$$

NOTE 128 – 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>.

iv) Otherwise, the result of

$$X \text{ <comp op> } Y$$

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) $X \text{ <comp op> } Y$

is false if and only if

$X < \text{comp op} > Y$

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 collating sequence indicated in Subclause 4.2.3, “Rules determining collating sequence usage”, based on 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 collating sequence *CS*.
 - d) Depending on the collating sequence, 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.

NOTE 129 – If the coercibility characteristic of the comparison is *Coercible*, then the collating sequence used is the default defined for the character repertoire. See also other Syntax Rules in this Subclause, Subclause 10.6, “<character set specification>”, and Subclause 11.30, “<character set definition>”.

- 4) The comparison of two binary string values, *X* and *Y*, is determined by comparison of their octets with the same ordinal position. If X_i and Y_i are the values of the *i*-th octets of *X* and *Y*, respectively, and if L_X is the length in octets of *X* AND L_Y is the length in octets of *Y*, then *X* is equal to *Y* if and only if $L_Y = L_X$ and if $X_i = Y_i$ for all *i*.
- 5) The comparison of two bit string values, *X* and *Y*, is determined by comparison of their bits with the same ordinal position. If X_i and Y_i are the values of the *i*-th bits of *X* and *Y*, respectively, and if L_X is the length in bits of *X* and L_Y is the length in bits of *Y*, then:
 - a) *X* is equal to *Y* if and only if $L_X = L_Y$ and $X_i = Y_i$ for all *i*.
 - b) *X* is less than *Y* if and only if:
 - i) $L_X < L_Y$ and $X_i = Y_i$ for all *i* less than or equal to L_X ; or
 - ii) $X_i = Y_i$ for all $i < n$ and $X_n = 0$ and $Y_n = 1$ for some *n* less than or equal to the minimum of L_X and L_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

$(X - Y) \text{ INTERVAL } H = \text{INTERVAL '0' } H$
is true .

b) X is less than Y if and only if

$(X - Y) \text{ INTERVAL } H < \text{INTERVAL '0' } H$
is true .

NOTE 130 – Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.7.1, “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 $A \text{ TO } B$ be the specified or implied datetime qualifier of X and $C \text{ TO } D$ 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>

$\text{CAST } (X \text{ AS INTERVAL } U)$

Let YVE be the <value expression>

$\text{CAST } (Y \text{ AS INTERVAL } U)$

a) X is equal to Y if and only if

$\text{CAST } (XVE \text{ AS INTEGER }) = \text{CAST } (YVE \text{ AS INTEGER })$
is true .

b) X is less than Y if and only if

$\text{CAST } (XVE \text{ AS INTEGER }) < \text{CAST } (YVE \text{ AS INTEGER })$
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 , $1 \text{ (one)} \leq i \leq L_X$, 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

- 1) Without Feature T042, “Extended LOB data type support”, no subfield of the declared row type of a <row value expression> that is simply contained in a <comparison predicate> shall be of declared type large object string.
- 2) Without Feature S024, “Enhanced structured types”, no subfield of the declared type of a <row value expression> that is simply contained in a <comparison predicate> shall be of a structured type.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

8.3 <between predicate>

Function

Specify a range comparison.

Format

```
<between predicate> ::=  
    <row value expression> [ NOT ] BETWEEN  
    [ ASYMMETRIC | SYMMETRIC ]  
    <row value expression> AND <row value expression>
```

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 expression>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 specify SYMMETRIC or ASYMMETRIC.
- 2) Without Feature S024, "Enhanced structured types", no subfield of the declared type of a <row value expression> that is simply contained in a <between predicate> shall be of a structured type.

8.4 <in predicate>**8.4 <in predicate>****Function**

Specify a quantified comparison.

Format

```

<in predicate> ::=
    <row value expression>
    [ 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) Let *IVL* be an <in value list>.

(*IVL*)

is equivalent to the <table value constructor>:

(VALUES *IVL*)
- 2) Let *RVC* be the <row value expression> and let *IPV* be the <in predicate value>.
- 3) The expression

RVC NOT IN *IPV*

is equivalent to

NOT (*RVC* IN *IPV*)
- 4) 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>.
- 2) Without Feature T042, “Extended LOB data type support”, no subfield of the declared row type of a <row value expression> or a <table subquery> contained in an <in predicate> shall be of declared type large object string.
- 3) Without Feature S024, “Enhanced structured types”, no subfield of the declared row type of a <row value expression> or a <table subquery> that is simply contained in an <in predicate> shall be of a structured type.
- 4) Without Feature S024, “Enhanced structured types”, no <value expression> simply contained in an <in value list> shall be of a structured type.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

8.5 <like predicate>

Function

Specify a pattern-match comparison.

Format

```
<like predicate> ::=  
    <character like predicate>  
    | <octet like predicate>  
  
<character like predicate> ::=  
    <character match value> [ NOT ] LIKE <character pattern>  
    [ ESCAPE <escape character> ]  
  
<character match value> ::= <character value expression>  
  
<character pattern> ::= <character value expression>  
  
<escape character> ::= <character value expression>  
  
<octet like predicate> ::=  
    <octet match value> [ NOT ] LIKE <octet pattern>  
    [ ESCAPE <escape octet> ]  
  
<octet match value> ::= <blob value expression>  
  
<octet pattern> ::= <blob value expression>  
  
<escape octet> ::= <blob value expression>
```

Syntax Rules

- 1) The declared types of <character match value>, <character pattern>, and <escape character> shall be character string. <character match value>, <character pattern>, and <escape character> shall be comparable.
- 2) The declared types of <octet match value>, <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 the <character match value>, 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) Case:

- i) If <escape character> is not specified, then the collating sequence used for the <like predicate> is determined by Table 3, "Collating sequence usage for comparisons", taking <character match value> as comparand 1 (one) and <character pattern> as comparand 2.
 - ii) Otherwise, let *C1* be the coercibility characteristic and collating sequence of the <character match value>, and *C2* be the coercibility characteristic and collating sequence of the <character pattern>. Let *C3* be the resulting coercibility characteristic and collating sequence as determined by Table 2, "Collating coercibility rules for dyadic operators", taking *C1* as the operand 1 (one) coercibility and *C2* as the operand 2 coercibility. The collating sequence used for the <like predicate> is determined by Table 3, "Collating sequence usage for comparisons", taking *C3* as the coercibility characteristic and collating sequence of comparand 1 (one) and <escape character> as comparand 2.
- 4) If <octet like predicate> is specified, then:
- a) Let *MB* be the <blob value expression> of the <octet match value>, let *PB* be the <blob value expression> of the <octet pattern>, and let *EB* be the <blob value expression> of the <escape octet> if one is specified.
 - b) *MB* NOT LIKE *PB*
is equivalent to
NOT (*MB* LIKE *PB*)
 - c) *MB* NOT LIKE *PB* ESCAPE *EB*
is equivalent to
NOT (*MB* LIKE *PB* ESCAPE *EB*)

Access Rules

None.

General Rules

- 1) Let *MCV* be the value of *MC* and let *PCV* be the value of *PC*. If *EC* is specified, then let *ECV* be its value.
- 2) Let *MBV* be the value of *MB* and let *PBV* be the value of *PB*. If *EB* is specified, then let *EBV* be its value.
- 3) If <character like predicate> is specified, then:
 - a) Case:
 - i) If ESCAPE is not specified and either *MCV* or *PCV* are null values, then the result of
MC LIKE *PC*

8.5 <like predicate>

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 131 – 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. 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.

- 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.
- c) The string *PCV* is a sequence of the minimum number of substring specifiers such that each <character representation> of *PCV* is part of exactly one substring specifier. A substring specifier is an arbitrary character specifier, an arbitrary string specifier, or any sequence of <character representation>s other than an arbitrary character specifier or an arbitrary string specifier.

d) 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 <character representation>s of *MCV* and each <character representation> of *MCV* is part of exactly one substring.
 - 2) If the *i*-th substring specifier of *PCV* is an arbitrary character specifier, the *i*-th substring of *MCV* is any single <character representation>.
 - 3) If the *i*-th substring specifier of *PCV* is an arbitrary string specifier, then the *i*-th substring of *MCV* is any sequence of 0 (zero) or more <character representation>s.
 - 4) If the *i*-th substring specifier of *PCV* is neither an arbitrary character specifier nor an arbitrary string specifier, then the *i*-th substring of *MCV* is equal to that substring specifier according to the collating sequence of the <like predicate>, without the appending of <space> characters to *MCV*, and has the same length as that substring specifier.
 - 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 132 – 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 a <percent> in the SQL_TEXT character repertoire.
 - c) <underscore> in the context of an <octet like predicate> has the same bit pattern as an <underscore> in the SQL_TEXT character repertoire.
 - 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*.

8.5 <like predicate>

- 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”, a <like predicate> shall not be an <octet like predicate>.
- 2) Without Feature F281, “LIKE enhancements”, the <character match value> shall be a column reference.
- 3) Without Feature F281, “LIKE enhancements”, a <character pattern> shall be a <value specification>.
- 4) Without Feature F281, “LIKE enhancements”, an <escape character> shall be a <value specification>.
- 5) Without Feature T042, “Extended LOB data type support”, a <character value expression> simply contained in a <like predicate> shall not be of declared type BINARY LARGE OBJECT or CHARACTER LARGE OBJECT
- 6) Without Feature F421, “National character”, and Feature T042, “Extended LOB data type support”, 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> ::=
    <character match value> [ 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> ::=
    <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> <colon> <regular character set identifier> <colon> <right bracket>

<character enumeration> ::=
    <character specifier>
    | <character specifier> <minus sign> <character specifier>

<regular character set identifier> ::= <identifier>
```

Syntax Rules

- 1) The declared types of <character match value>, <similar pattern>, and <escape character> shall be character string. <character match value>, <similar pattern>, and <escape character> shall be comparable.
- 2) Let *CM* be the <character match value> 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, or ALNUM.
- 4) Case:
 - a) If <escape character> is not specified, then the collating sequence used for the <similar predicate> is determined by Table 3, "Collating sequence usage for comparisons", taking <character match value> as comparand 1 (one) and <similar pattern> as comparand 2.
 - b) Otherwise, let *C1* be the coercibility characteristic and collating sequence of the <character match value>, and *C2* be the coercibility characteristic and collating sequence of the <similar pattern>. Let *C3* be the resulting coercibility characteristic and collating sequence as determined by Table 2, "Collating coercibility rules for dyadic operators", taking *C1* as the operand 1 (one) coercibility and *C2* as the operand 2 coercibility. The collating sequence used for the <similar predicate> is determined by Table 3, "Collating sequence usage for comparisons", taking *C3* as the coercibility characteristic and collating sequence of comparand 1 (one) and <escape character> as comparand 2.

It is implementation-defined, whether all, some, or no collating sequences other than the default collating sequence for the character set of the <character match value> can be used as the collating sequence of the <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>, 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>, 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.

8.6 <similar predicate>

- 7) A <character enumeration> shall not be specified in a way that both its first and its last <character specifier>s are <non-escaped character>s that are <colon>s.

Access Rules

None.

General Rules

- 1) Let *MCV* be the result of the <character value expression> of the <character match value> 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: If the length in characters of *ECV* is not equal to 1 (one), then an exception condition is raised: *data exception — invalid escape character*.
 - a) If *ECV* is one of <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore> or <percent> 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*.
 - b) 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 133 – 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 collating sequence of the pattern *P*.
 - b) Otherwise, the character that the <character specifier> in the <character enumeration> represents.

- 6) 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) $L(Q1 \text{ <vertical bar> } Q2)$
is the union of $L(Q1)$ and $L(Q2)$
 - b) $L(Q \text{ <asterisk> })$
is the set of all strings that can be constructed by concatenating zero or more strings from $L(Q)$.
 - c) $L(Q \text{ <plus sign> })$
is the set of all strings that can be constructed by concatenating one or more strings from $L(Q)$.
 - d) $L(\text{<character specifier>})$
is a set that contains a single string of length 1 (one) with the character that the <character specifier> represents
 - e) $L(\text{<percent>})$
is the set of all strings of any length (zero or more) from the character set of the pattern P .
 - f) $L(\text{<left paren> } Q \text{ <right paren> })$
is equal to $L(Q)$
 - g) $L(\text{<underscore>})$
is the set of all strings of length 1 (one) from the character set of the pattern P .
 - h) $L(\text{<left bracket> <character enumeration> <right bracket>})$
is the set of all strings of length 1 (one) from the set of characters in the <character enumeration>s.
 - i) $L(\text{<left bracket> <circumflex> <character enumeration> <right bracket>})$
is the set of all strings of length 1 (one) with characters from the character set of the pattern P that are not contained in the set of characters in the <character enumeration>.
 - j) $L(\text{<left bracket> <colon> ALPHA <colon> <right bracket>})$
is the set of all character strings of length 1 (one) that are <simple Latin letter>s.
 - k) $L(\text{<left bracket> <colon> UPPER <colon> <right bracket>})$
is the set of all character strings of length 1 (one) that are <simple Latin upper case letter>s.
 - l) $L(\text{<left bracket> <colon> LOWER <colon> <right bracket>})$
is the set of all character strings of length 1 (one) that are <simple Latin lower case letter>s.
 - m) $L(\text{<left bracket> <colon> DIGIT <colon> <right bracket>})$
is the set of all character strings of length 1 (one) that are <digit>s.

8.6 <similar predicate>

n) $L(\langle \text{left bracket} \rangle \langle \text{colon} \rangle \text{ALNUM} \langle \text{colon} \rangle \langle \text{right bracket} \rangle)$

is the set of all character strings of length 1 (one) that are <simple Latin letter>s or <digit>s.

o) $L(Q1 \parallel Q2)$

is the set of all strings that can be constructed by concatenating one element of $L(Q1)$ and one element of $L(Q2)$.

p) $L(Q)$

is the set of the zero-length string, if Q is an empty regular expression.

7) The <similar predicate>

$CM \text{ SIMILAR TO } SP$

is true, if there exists at least one element X of $L(R)$ that is equal to MCV according to the collating sequence of the <similar predicate>; otherwise, it is false.

NOTE 134 – The <similar predicate> is defined differently from equivalent forms of the LIKE predicate. In particular, blanks at the end of a pattern and collating sequences are handled differently.

Conformance Rules

- 1) Without Feature T141, “SIMILAR predicate”, conforming SQL language shall contain no <similar predicate>.

8.7 <null predicate>

Function

Specify a test for a null value.

Format

<null predicate> ::=
 <row value expression> IS [NOT] NULL

Syntax Rules

None.

Access Rules

None.

General Rules

- 1) Let R be the value of the <row value expression>.
- 2) If every value in R is the null value, then “ R IS NULL” is true; otherwise, it is false.
- 3) If no value in R is the null value, then “ R IS NOT NULL” is true; otherwise, it is false.

NOTE 135 – 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 16, “<null predicate> semantics”, specifies this behavior.

Table 16—<null predicate> semantics

Expression	R IS NULL	R IS NOT NULL	NOT R IS NULL	NOT R IS NOT NULL
degree 1: null	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>
degree 1: not null	<u>false</u>	<u>true</u>	<u>true</u>	<u>false</u>
degree > 1: all null	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>
degree > 1: some null	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>
degree > 1: none null	<u>false</u>	<u>true</u>	<u>true</u>	<u>false</u>

Conformance Rules

None.

8.8 <quantified comparison predicate>

Function

Specify a quantified comparison.

Format

<quantified comparison predicate> ::=
 <row value expression> <comp op> <quantifier> <table subquery>

<quantifier> ::= <all> | <some>

<all> ::= ALL

<some> ::= SOME | ANY

Syntax Rules

- 1) Let *RV1* and *RV2* be <value expression>s whose declared types are respectively that of the <row value expression> 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 expression> 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

- 1) Without Feature T042, “Extended LOB data type support”, no subfield of the declared row type of a <row value expression> or a <table subquery> contained in a <quantified comparison predicate> shall be of declared type large object string.
- 2) Without Feature S024, “Enhanced structured types”, no subfield of the declared row type of a <row value expression> shall be of a structured type.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

8.9 <exists predicate>

Function

Specify a test for a non-empty set.

Format

```
<exists predicate> ::=  
    EXISTS <table subquery>
```

Syntax Rules

None.

Access Rules

None.

General Rules

- 1) Let T be the result of the <table subquery>.
- 2) If the cardinality of T is greater than 0 (zero), then the result of the <exists predicate> is true; otherwise, the result of the <exists predicate> is false.

Conformance Rules

None.

8.10 <unique predicate>

Function

Specify a test for the absence of duplicate rows.

Format

```
<unique predicate> ::=  
    UNIQUE <table subquery>
```

Syntax Rules

None.

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 equal to the value of the corresponding column in the other row according to Subclause 8.2, "<comparison predicate>", 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 any <unique predicate>.
- 2) Without Feature F291, "UNIQUE predicate" and Feature S024, "Enhanced structured types", no column of the result of the <table subquery> shall be of structured type.

8.11 <match predicate>

Function

Specify a test for matching rows.

Format

```
<match predicate> ::=  
    <row value expression> MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ]  
    <table subquery>
```

Syntax Rules

- 1) The row type of the <row value constructor> and the row type of the <table subquery> shall be comparable.
- 2) The collating sequence for each pair of respective values in the <match predicate> is determined in the same manner as described in Subclause 8.2, "<comparison predicate>".
- 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 expression>.
- 2) If SIMPLE is specified or implicit, then

Case:

- a) If some value in R is the null value, then the <match predicate> is true.
- b) If no value in R is the null value, then

Case:

- i) If UNIQUE is not specified and there exists a (possibly non-unique) row RT_i of the <table subquery> such that

$$R = RT_i$$

then the <match predicate> is true.

- ii) If UNIQUE is specified and there is a unique row RT_i of the <table subquery> such that

$$R = RT_i$$

then the <match predicate> is true.

- iii) Otherwise, the <match predicate> is false.

3) If PARTIAL is specified, then

Case:

- a) If all values in R are the null value, then the <match predicate> is true.
- b) Otherwise,

Case:

- i) If UNIQUE is not specified and there exists a (possibly non-unique) 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.
- ii) If UNIQUE is specified and there is a unique 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.
- iii) Otherwise, the <match predicate> is false.

4) If FULL is specified, then

Case:

- a) If all values in R are the null value, then the <match predicate> is true.
- b) If no values in R are the null value, then

Case:

- i) If UNIQUE is not specified and there exists a (possibly non-unique) row RT_i of the <table subquery> such that

$$R = RT_i$$
 then the <match predicate> is true.
- ii) If UNIQUE is specified and there exists a unique row RT_i of the <table subquery> such that

$$R = RT_i$$
 then the <match predicate> is true.
- iii) Otherwise, the <match predicate> is false.
- c) Otherwise, the <match predicate> is false.

Conformance Rules

- 1) Without Feature F741, "Referential MATCH types", conforming SQL language shall not contain any <match predicate>.
- 2) Without Feature F741, "Referential MATCH types", and Feature S024, "Enhanced structured types", no subfield of the declared row type of the <row value expression> shall be of a structured type and no column of the result of the <table subquery> shall be of a structured type.

8.12 <overlaps predicate>

Function

Specify a test for an overlap between two datetime periods.

Format

<overlaps predicate> ::=
 <row value expression 1> OVERLAPS <row value expression 2>

<row value expression 1> ::= <row value expression>

<row value expression 2> ::= <row value expression>

Syntax Rules

- 1) The degrees of <row value expression 1> and <row value expression 2> shall both be 2.
- 2) The declared types of the first field of <row value expression 1> and the first field of <row value expression 2> shall both be datetime data types and these data types shall be comparable.
NOTE 136 – Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.7.1, “Datetimes”.
- 3) The declared type of the second field of each <row value expression> 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 expression>.
 - 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 expression>.

Access Rules

None

General Rules

- 1) Let $D1$ be the value of the first field of <row value expression 1> and $D2$ be the value of the first field of <row value expression 2>.
- 2) Case:
 - a) If the most specific type of the second field of <row value expression 1> is a datetime data type, then let $E1$ be the value of the second field of <row value expression 1>.
 - b) If the most specific type of the second field of <row value expression 1> is INTERVAL, then let $I1$ be the value of the second field of <row value expression 1>. Let $E1 = D1 + I1$.

- 3) If $D1$ is the null value or if $E1 < D1$, then let $S1 = E1$ and let $T1 = D1$. Otherwise, let $S1 = D1$ and let $T1 = E1$.
- 4) Case:
 - a) If the most specific type of the second field of <row value expression 2> is a datetime data type, then let $E2$ be the value of the second field of <row value expression 2>.
 - b) If the most specific type of the second field of <row value expression 2> is INTERVAL, then let $I2$ be the value of the second field of <row value expression 2>. Let $E2 = D2 + I2$.
- 5) If $D2$ is the null value or if $E2 < D2$, then let $S2 = E2$ and let $T2 = D2$. Otherwise, let $S2 = D2$ and let $T2 = E2$.
- 6) 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 F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain any <overlaps predicate>.

8.13 <distinct predicate>

Function

Specify a test of whether two row values are distinct

Format

```
<distinct predicate> ::=  
    <row value expression 3> IS DISTINCT FROM <row value expression 4>  
  
<row value expression 3> ::= <row value expression>  
  
<row value expression 4> ::= <row value expression>
```

Syntax Rules

- 1) The two <row value expression>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 expression>s shall be comparable.
- 4) Let *X* be a value in the first <row value expression> and let *Y* be the respective value in the second <row value expression>.

Access Rules

None.

General Rules

- 1) The result of the <distinct predicate> is either true or false.
- 2) Two <row value expression>s are distinct if, for any pair of respective values *X* and *Y*, *X IS DISTINCT FROM Y* is true.
- 3) Case:
 - a) If the declared type of *X* or *Y* is an array type, then “*X IS DISTINCT FROM Y*” is effectively computed as follows:
 - i) Let *N_X* be the number of elements in *X*; let *N_Y* be the number of elements in *Y*.
 - ii) Let *EX_i* be the *i*-th element of *X*; let *EY_i* be the *i*-th element of *Y*.
 - iii) Case:
 - 1) If *N_X* is not equal to *N_Y*, then “*X IS DISTINCT FROM Y*” is true.
 - 2) If *N_X* equals zero and *N_Y* equals zero, then “*X IS DISTINCT FROM Y*” is false.

- 3) If “ EX_i IS DISTINCT FROM EY_i ” is false for all i between 1 (one) and NX , then “ X IS DISTINCT FROM Y ” is false.
 - 4) Otherwise, “ X IS DISTINCT FROM Y ” is true.
- b) Otherwise,
- Case:
- i) “ X IS DISTINCT FROM Y ” is false if either:
 - 1) X and Y are the null value, or
 - 2) $X = Y$ according to Subclause 8.2, “<comparison predicate>”.
 - ii) Otherwise, “ X IS DISTINCT FROM Y ” is true.
- 4) If two <row value expression>s are not distinct, then they are said to be *duplicates*. If a number of <row value expression>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 specify any <distinct predicate>.
- 2) Without Feature T151, “DISTINCT predicate”, and Feature S024, “Enhanced structured types”, no subfield of the declared row type of either <row value expression> shall be of a structured type.

8.14 <type predicate>

Function

Specify a type test.

Format

```
<type predicate> ::=
    <user-defined type value expression> 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> ::=
    <user-defined type>

<exclusive user-defined type specification> ::=
    ONLY <user-defined type>
```

Syntax Rules

- 1) 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*.
- 2) 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*.
- 3) If *T1* and *T2* are specified types, then *T1* shall be a member of the subtype family of *T2*.
NOTE 137 – The term “subtype family” is defined in Subclause 4.8.3, “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.
- 4) Let *UVE* be the <user-defined type value expression>. Let *TL* be the <type list>.
- 5) A <type predicate> of the form
 UVE IS NOT OF (*TL*)
is equivalent to
 NOT (*UVE* IS OF (*TL*))

Access Rules

None.

General Rules

- 1) Let V be the result of evaluating the <user-defined type value expression>.
- 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>.

8.15 <search condition>

Function

Specify a condition that is true, false, or unknown, depending on the value of a <boolean value expression>.

Format

```
<search condition> ::=  
    <boolean value expression>
```

Syntax Rules

None.

Access Rules

None.

General Rules

- 1) When a <search condition> *S* is evaluated against a row of a table, each reference to a column of that table by a column reference directly contained in *S* is a reference to the value of that column in that row.
- 2) The result of the <search condition> is the result of the <boolean value expression>.

Conformance Rules

None.

9 Data assignment rules and routine determination

9.1 Retrieval assignment

Function

Specify rules for assignments that to targets that do not support null values or that support null values with indicator parameters (*e.g.*, assigning SQL-data to host parameters).

Syntax Rules

- 1) Let T and V be a *TARGET* and *VALUE* specified in an application of this Subclause.
- 2) If the declared type of T is bit string, binary string, numeric, boolean, datetime, interval, or a user-defined type, then either the declared type of V shall be a mutually assignable bit string type, a binary string type, a numeric type, a boolean type, a mutually assignable datetime type, a mutually assignable interval type, or a subtype of the declared type of T , respectively, or an appropriate user-defined cast function *UDCF* shall be available to assign V to T .
 NOTE 138 – “Appropriate user-defined cast function” is defined in Subclause 4.13, “Data conversions”.
 NOTE 139 – “subtype” is defined in Subclause 4.8.3, “Subtypes and supertypes”.
- 3) If the declared type of T is character string, then
 Case:
 - a) If T is either a locator parameter of an external routine or a host parameter that is a character large object locator parameter, then the declared type of V shall be character large object type and the character string type of T and the declared type of V shall be mutually assignable.
 - b) Otherwise, the declared type of V shall be a mutually assignable character string type.
- 4) 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 .
- 5) 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 , $1 \text{ (one)} \leq i \leq n$, 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 , $1 \text{ (one)} \leq i \leq n$, the Syntax Rules of this Subclause apply to $T_i V_i$, as *TARGET* and *VALUE*, respectively.
- 6) If the declared type of T is an array type, then:
 - a) The declared type of V shall be an array type.

- b) 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 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, bit string, or binary string and the length M in characters, bits, or octets, respectively, of V is greater than the length in characters, bits, 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).
- 4) 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 character large object 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 .
- g) If the declared type of T is character large object 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 , 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 bit string with length in bits L and the length in bits 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 bit string with length in bits L and the length in bits of V is greater than L , then the value of T is set to the first L bits of V and a completion condition is raised: *warning — string data, right truncation*.
- j) If the declared type of T is fixed-length bit string with length in bits L and the length in bits M of V is smaller than L , then the first M bits of T are set to V , the remaining bits of T are set to bits each with the value of 0 (zero), and a completion condition is raised: *warning — implicit zero-bit padding*.
- k) If the declared type of T is variable-length bit string and the length in bits M of V is not greater than the maximum length in bits of T , then the value of T is set to V and the length in bits of T is set to M .
- l) If the declared type of T is variable-length bit string, and the length in bits of V is greater than the maximum length in bits L of T , then the value of T is set to the first L bits of V , the length in bits of T is set to L , and a completion condition is raised: *warning — string data, right truncation*.
- m) If the declared type of T is 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 .
- n) If the declared type of T is 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 , 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* TO *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 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 140 – The maximum cardinality L of T is unchanged.

Conformance Rules

None.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

9.2 Store assignment

Function

Specify rules for assignments where the target permits null without the use of indicator parameters, such as storing SQL-data or setting the value of SQL parameters.

Syntax Rules

- 1) Let T and V be a *TARGET* and *VALUE* specified in an application of this Subclause.
- 2) If the declared type of T is character string, bit string, binary string, numeric, boolean, datetime, interval, or a user-defined type, then either the declared type of V shall be a mutually assignable character string type, a mutually assignable bit string type, a binary string type, a numeric type, a boolean type, a mutually assignable datetime type, a mutually assignable interval type, or a subtype of the declared type of T , respectively, or an appropriate user-defined cast function *UDCF* shall be available to assign V to T .
NOTE 141 – “Appropriate user-defined cast function” is defined in Subclause 4.13, “Data conversions”.
NOTE 142 – “subtype” is defined in Subclause 4.8.3, “Subtypes and supertypes”.
- 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 , $1 \text{ (one)} \leq i \leq n$, 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 , $1 \text{ (one)} \leq i \leq n$, the Syntax Rules of this Subclause apply to $T1_i$ $V1_i$, as *TARGET* and *VALUE*, respectively.
- 5) If the declared type of T is an array type, then:
 - a) The declared type of V shall be an array type.
 - b) 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) 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*.

9.2 Store assignment

- vi) If the declared type of T is character large object 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 .
- vii) If the declared type of T is character large object 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*.
- viii) If the declared type of T is fixed-length bit string with length in bits L and the length in bits 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 bit string with length in bits L and the length in bits M of V is greater than L , then an exception condition is raised: *data exception — string data, right truncation*.
- x) If the declared type of T is fixed-length bit string with length in bits L and the length in bits M of V is less than L , then an exception condition is raised: *data exception — string data, length mismatch*.
- xi) If the declared type of T is variable-length bit string and the length in bits M of V is not greater than the maximum length in bits of T , then the value of T is set to V and the length in bits of T is set to M .
- xii) If the declared type of T is variable-length bit string, and the length in bits M of V is greater than the maximum length in bits L of T , then an exception condition is raised: *data exception — string data, right truncation*.
- xiii) If the declared type of T is 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 .
- xiv) If the declared type of T is 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*.
- 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

$$\text{CAST} (V \text{ TO } DT)$$
 - 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 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*.

9.2 Store assignment

- 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 null, 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 143 – The maximum cardinality L of T is unchanged.

Conformance Rules

None.

9.3 Data types of results of aggregations

Function

Specify the result data type of the result of an aggregation over 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 *DTS* be a set of data types specified in an application of this Subclause.
- 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 all data types in *DTS* shall be character string, and all of them shall have the same character repertoire. That character repertoire is the character repertoire of the result. The character set of the result is the character set of one of the data types in *DTS*. The specific character set chosen is implementation-dependent. The collating sequence and the coercibility characteristic are determined as specified in Table 2, "Collating coercibility rules for dyadic operators".

Case:

- i) If any of the data types in *DTS* is character large object string, then the result data type is character large object 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*.
 - ii) 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*.
 - iii) 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 bit string, then all data types in *DTS* shall be bit string.

Case:

- i) If any of the data types in *DTS* is variable-length bit string, then the result data type is variable-length bit string with maximum length in bits equal to the maximum of the lengths in bits and maximum lengths in bits of the data types in *DTS*.
 - ii) Otherwise, the result data type is fixed-length bit string with length in bits equal to the maximum of the lengths in bits of the data types in *DTS*.
- c) If any of the data types in *DTS* is binary string, then the result data type is binary string with maximum length in octets equal to the maximum of the lengths in octets and maximum lengths in octets of the data types in *DTS*.
- d) 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*.

9.3 Data types of results of aggregations

- e) 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.
- f) 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 timezone displacement value, then the result has a timezone displacement value; otherwise, the result does not have a timezone displacement value.
- g) 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.
- h) If any data type in *DTS* is boolean, then each data type in *DTS* shall be boolean. The result data type is boolean.
- i) 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 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.
- j) 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*.
- 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 a <scope table name list> and every table name that appears in some such <scope table name list> appears in every such <scope table name list>, then let *STNL* be a <scope table name list> consisting of every such table name in some order, arbitrarily chosen. The result data type is:

RT SCOPE(*STNL*)

- 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 144 – *Minimal common supertype* is defined in Subclause 4.8.3, “Subtypes and supertypes”.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

9.4 Subject routine determination

Function

Determine the subject routine of a given routine invocation.

Syntax Rules

- 1) Let SR and AL be respectively a 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 , $1 \text{ (one)} \leq i \leq n$, 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 , $1 \text{ (one)} \leq j \leq m$, be the j -th SQL argument in AL .
- 4) Let $SDTA_j$ be the declared type of A_j .
- 5) Let $SDTP_{ij}$ 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 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 $SDTP_{p,r} \prec SDTP_{q,r}$ in the type precedence list of $SDTA_r$, then eliminate R_q from SR .
NOTE 145 – 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

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 *data type names* as specified in the Syntax Rules of this Subclause.
- 3) Let " $A \prec B$ " represent " A has precedence over B " and let " $A \simeq B$ " represent " A has the same precedence as B ".
- 4) If DT is a user-defined type, then let DTN be the type designator of DT .

NOTE 146 – The type designator of a user-defined type is the type name included in its user-defined type descriptor.

Case:

- a) If DT is a maximal supertype, then TPL is

DTN

- b) Otherwise:

- i) Let ST be the set of supertypes of DT . Let n be the number of data types in ST .
- ii) For each data type T in ST , the following set P of precedence relationships $\prec$ is valid:
 - 1) For all direct supertypes T_i of T , $T \prec T_i$.
 - 2) For all direct supertypes T_i and T_j of T , if T_i occurs immediately prior to T_j in the <subtype clause> of <user-defined type body> of T , then $T_i \prec T_j$.

- iii) TPL is constructed as follows:

- 1) TPL is initially empty.
- 2) For i ranging from 1 (one) to n :

- A) Let NT be the set of types in ST such that no other type T_j in $ST \prec$ a type T_k in NT according to P .

- B) Case:

- I) If there is exactly one type T_k in NT , then the type designator of T_k is placed next in TPL and all relationships of the form " $T_k \prec T_r$ " are removed from P , where T_r is any other type in ST .
- II) If there is more than one type T_k in NT , then the type T_k that has a direct subtype that occurs closest to the tail of the portion of TPL constructed so far is placed next in TPL and all relationships of the form " $T_k \prec T_r$ " are removed from P , where T_r is any other type in ST .

- III) Otherwise, TPL is empty.

9.5 Type precedence list determination

- 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 binary string, then *TPL* is
BINARY LARGE OBJECT
- 8) If *DT* is fixed-length bit string, then *TPL* is
BIT, BIT VARYING
- 9) If *DT* is variable-length bit character string, then *TPL* is
BIT VARYING
- 10) If *DT* is numeric, then:
 - a) Let *NDT* be the following set of numeric types: SMALLINT, INTEGER, NUMERIC, DECIMAL, REAL, FLOAT, and DOUBLE PRECISION.
If the radix of any of the exact numeric types SMALLINT, INTEGER, NUMERIC, or DECIMAL is decimal, then let “precision” mean the product of $\log_2(10)$ and the actual specified or implementation-defined precision.
 - b) Let T_i and T_j be valid data types in *NDT*. The following set *P* of precedence relationships between T_i and T_j shall be valid:
 - i) If the implementation-defined precision of INTEGER is greater than the implementation-defined precision of SMALLINT, then $\text{SMALLINT} \prec \text{INTEGER}$; otherwise, $\text{SMALLINT} \simeq \text{INTEGER}$.
 - ii) Case:
 - 1) If the implementation-defined maximum precision of DECIMAL is greater than the implementation-defined precision of INTEGER, then $\text{INTEGER} \prec \text{DECIMAL}$.
 - 2) If the implementation-defined maximum precision of DECIMAL is equal to the implementation-defined precision of INTEGER, then $\text{INTEGER} \simeq \text{DECIMAL}$.
 - 3) Otherwise, $\text{DECIMAL} \prec \text{INTEGER}$ and
Case:
 - A) If the implementation-defined maximum precision of DECIMAL is greater than the implementation-defined precision of SMALLINT, then $\text{SMALLINT} \prec \text{DECIMAL}$,
 - B) If the implementation-defined maximum precision of SMALLINT is greater than the implementation-defined precision of DECIMAL, then $\text{DECIMAL} \prec \text{SMALLINT}$.
 - C) Otherwise, $\text{SMALLINT} \simeq \text{DECIMAL}$.

9.5 Type precedence list determination

iii) Case:

- 1) If the implementation-defined maximum precision of NUMERIC is greater than the implementation-defined precision of INTEGER, then $\text{INTEGER} < \text{NUMERIC}$.
- 2) If the implementation-defined maximum precision of NUMERIC is equal to the implementation-defined precision of INTEGER, then $\text{INTEGER} \simeq \text{NUMERIC}$.
- 3) Otherwise, $\text{NUMERIC} < \text{INTEGER}$ and

Case:

- A) If the implementation-defined maximum precision of NUMERIC is greater than the implementation-defined precision of SMALLINT, then $\text{SMALLINT} < \text{NUMERIC}$.
- B) If the implementation-defined precision of SMALLINT is greater than the implementation-defined maximum precision of NUMERIC, then $\text{NUMERIC} < \text{SMALLINT}$.
- C) Otherwise, $\text{SMALLINT} \simeq \text{NUMERIC}$.

iv) Case:

- 1) If the implementation-defined maximum precision of NUMERIC is greater than the implementation-defined maximum precision of DECIMAL, then $\text{DECIMAL} < \text{NUMERIC}$.
- 2) If the implementation-defined maximum precision of NUMERIC is equal to the implementation-defined maximum precision of DECIMAL, then $\text{DECIMAL} \simeq \text{NUMERIC}$.
- 3) Otherwise, $\text{NUMERIC} < \text{DECIMAL}$.

v) $\text{REAL} < \text{DOUBLE PRECISION}$

vi) Case:

- 1) If the implementation-defined maximum precision of FLOAT is greater than the implementation-defined precision of REAL, then $\text{REAL} < \text{FLOAT}$.
- 2) If the implementation-defined maximum precision of FLOAT is equal to the implementation-defined precision of REAL, then $\text{FLOAT} \simeq \text{REAL}$.
- 3) Otherwise, $\text{FLOAT} < \text{REAL}$.

vii) Case:

- 1) If the implementation-defined maximum precision of FLOAT is greater than the implementation-defined precision of DOUBLE PRECISION, then $\text{DOUBLE PRECISION} < \text{FLOAT}$.
- 2) If the implementation-defined maximum precision of FLOAT is equal to the implementation-defined precision of DOUBLE PRECISION, then $\text{FLOAT} \simeq \text{DOUBLE PRECISION}$.
- 3) Otherwise, $\text{FLOAT} < \text{DOUBLE PRECISION}$.

9.5 Type precedence list determination

- viii) Let *MAXEX* be whichever of INTEGER, NUMERIC, and DECIMAL has the greatest precedence and let *MINAP* be whichever of REAL and FLOAT has the least precedence. $MAXEX \prec MINAP$.
- c) Let *PTC* be the transitive closure of *P*.
- d) *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 $DT \prec T$ or $DT \simeq T$ 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 $T_j \prec T_k$ 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 $T_k \prec T_r$ 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 $T_k \prec T_r$, $T_k \simeq T_r$, $T_s \prec T_r$, and $T_s \simeq T_r$ are removed from *PTC*, where T_r is any type in *ST*.
- 11) If *DT* specifies a year-month interval type, then *TPL* is
INTERVAL YEAR
- 12) If *DT* specifies a day-time interval type, then *TPL* is
INTERVAL DAY
- 13) If *DT* specifies DATE, then *TPL* is
DATE
- 14) If *DT* specifies TIME, then *TPL* is
TIME
- 15) If *DT* specifies TIMESTAMP, then *TPL* is
TIMESTAMP
- 16) If *DT* specifies BOOLEAN, then *TPL* is
BOOLEAN
- 17) If *DT* is a collection type, then let *CTC* be the <collection type constructor> 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
 $RIO_1 \text{ CTC}, RIO_2 \text{ CTC}, \dots, RIO_n \text{ CTC}$

9.5 Type precedence list determination

- 18) 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

KAW₁, KAW₂, . . . , KAW_n

- 19) If *DT* is a row type, then *TPL* is

ROW

NOTE 147 – 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

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.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

9.7 Type name determination

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 LARGE OBJECT, then let *FNSDT* be "BLOB".
- e) If *DT* specifies BIT, then let *FNSDT* be "BIT".
- f) If *DT* specifies BIT VARYING, then let *FNSDT* be "BITVAR".
- g) If *DT* specifies SMALLINT, then let *FNSDT* be "SMALLINT".
- h) If *DT* specifies INTEGER, then let *FNSDT* be "INTEGER".
- i) If *DT* specifies DECIMAL, then let *FNSDT* be "DECIMAL".
- j) If *DT* specifies NUMERIC, then let *FNSDT* be "NUMERIC".
- k) If *DT* specifies REAL, then let *FNSDT* be "REAL".
- l) If *DT* specifies FLOAT, then let *FNSDT* be "FLOAT".
- m) If *DT* specifies DOUBLE PRECISION, then let *FNSDT* be "DOUBLE".
- n) If *DT* specifies DATE, then let *FNSDT* be "DATE".
- o) If *DT* specifies TIME, then let *FNSDT* be "TIME".
- p) If *DT* specifies TIMESTAMP, then let *FNSDT* be "TIMESTAMP".
- q) If *DT* specifies INTERVAL, then let *FNSDT* be "INTERVAL".

Access Rules

None.

General Rules

None.

Conformance Rules

None.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

10 Additional common elements

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.

10.1 <interval qualifier>

- 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 date-time 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 (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 <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 (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 any <interval qualifier>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

10.2 <language clause>

Function

Specify a standard programming language.

Format

```
<language clause> ::=  
    LANGUAGE <language name>
```

```
<language name> ::=  
    ADA | C | COBOL | FORTRAN | MUMPS | PASCAL | PLI | SQL
```

Syntax Rules

None.

Access Rules

None.

General Rules

- 1) The standard programming language specified by the clause is defined in the International Standard identified by the <language name> keyword. Table 17, "Standard programming languages", specifies the relationship.

Table 17—Standard programming languages

Language keyword	Relevant standard
ADA	ISO/IEC 8652
C	ISO/IEC 9899
COBOL	ISO 1989
FORTRAN	ISO 1539
MUMPS	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.
- 2) If no <schema name> in <schema name list> *SO* contains the <qualified identifier> INFORMATION_SCHEMA, then *SO* is effectively replaced by
INFORMATION_SCHEMA, *SO*

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 any <path specification>.

10.4 <routine invocation>

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 <user-defined type>
```

Syntax Rules

- 1) Let *RI* be the <routine invocation>, let *TP* be the SQL-path (if any), and let *UDTSM* be the user-defined type of the static SQL-invoked method (if any) specified in an application of this Subclause.
- 2) Let *RN* be the <routine name> immediately contained in the <routine invocation> *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 invocation>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an 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* does not include a STATIC indication.
 - c) If *RI* is immediately contained in a <static method invocation>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an 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 *MSD* for *R* includes a STATIC indication and *MSD* is included in a user-defined type descriptor for *UDTSM* or for some supertype of *UDTSM*.

- d) Otherwise, an SQL-invoked routine R is a *possibly candidate routine* for RI if R is an SQL-invoked function that is not an SQL-invoked method 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 of 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 .

NOTE 148 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

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 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 RI that is an executable routine such that:

- 1) RI has NA SQL parameters.
- 2) If RI is not immediately contained in a <call statement>, then for each P_i ,

Case:

- A) If the declared type of P_i is a user-defined data 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 149 – “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 150 – “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 <static 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 *R1* such that:

A) *R1* is an invocable routine.

B) The <schema name> of the schema of *R1* is in *DP*.

C) Case:

I) If the routine descriptor of *R1* 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 *R1*.

II) If the routine descriptor of *R1* includes a STATIC indication, then there is no other invocable routine *R2* for which the user-defined type described by the user-defined 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 *R1*.

ii) If *RN* contains a <schema name> *SN*, then

Case:

1) If *SN* is "INFORMATION_SCHEMA", then the single candidate routine of *RI* is the built-in function identified by <routine name>.

2) Otherwise, *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 <user-defined type> 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> or a <static 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
Case:
 - 1) If *SN* is "INFORMATION_SCHEMA", then the single candidate routine of *RI* is the built-in function identified by <routine name>.
 - 2) Otherwise, *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 *R1* 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 *R1*.
- 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 *A_i* is 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.
 - B) If *A_i* is the <SQL parameter name> of an SQL parameter of an SQL-invoked routine, 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.
- 5) For each *P_i* that is an input SQL parameter but not an output parameter, *A_i* shall be a <value expression> or <generalized 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 <user-defined type> 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) For each *A_i*, let *V_i* be a value arbitrarily chosen whose declared type is *TS_i*. Let *XAL* be an <SQL argument list> with *N* <SQL argument>s derived from values *V_i* 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*.

Case:

- A) 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*.

B) Otherwise:

I) There shall be at least one candidate subject routine in *CSR*.

II) Case:

- 1) If there is exactly one candidate subject routine in *CSR*, then the subject routine *SR* is the candidate subject routine in *CSR*.
- 2) If there is more than one candidate subject routine in *CSR*, then

Case:

- a) If *RI* is not immediately contained in a <static method selection>, then the subject routine *SR* is an SQL-invoked routine *R1* in *CSR* such that there is no other candidate subject routine *R2* in *CSR* for which the <schema name> of the schema that includes *R2* precedes in *DP* the <schema name> of the schema that includes *R1*.
- b) Otherwise, the subject routine *SR* is an SQL-invoked routine *R1* 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 *R1*.

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 and A_i as *TARGET* and *VALUE*, respectively.

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 immediately contained in a <new invocation>.

Access Rules

None.

General Rules

- 1) Let *SAL* and *SR* be the static SQL argument list and subject routine of the <routine invocation> as specified in an application of this Subclause.
NOTE 151 – “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) Case:
 - 1) If *SR* is an instance SQL-invoked method, then:
 - A) 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) $\leq i \leq N$, be the *i*-th SQL parameter of *SR* and PM_i , 1 (one) $\leq i \leq N$, 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) The declared type of PM_j , $2 \leq j \leq N$, is compatible with the declared type of PSR_j .

NOTE 152 – *SR* is an element of the set *SM*.
 - B) *SM* is the set of overriding methods of *SR* and every SQL-invoked method *M* in *SM* is an overriding method of *SR*.
 - C) 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*; otherwise, let *DT1* be the most specific type of the first value $V1$ in *DAL*.
 - D) 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 of the assignment of V_i to a target of type T_i according to the rules of Subclause 9.2, "Store assignment".
 - b) Otherwise, let CPV_i be an implementation-defined value of declared type T_i .
- 4) If R is a built-in function BIF , then

Case:

- a) If the syntax for invoking BIF is defined in ISO/IEC 9075, then the result of <routine invocation> is as defined for that syntax in ISO/IEC 9075.
 - b) Otherwise, the result of <routine invocation> is implementation-defined.
- 5) 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 .

- 6) 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 character set name substitution value, the SQL-path of the current SQL-session, the current default time zone, 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 transaction access mode, the current transaction isolation level, and the current transaction diagnostics area limit to their values in *CSC*.
 - c) 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 position of all open 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 position of all open cursors that are defined by <declare cursor>s that are contained in <SQL-client module definition>s that are not the <SQL-client module definition> 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 position of all open cursors that are defined by <declare cursor>s that are contained in the <SQL-client module definition> of *P* are removed from *RSC*.
 - d) Indicate in the routine execution context of *RSC* that the SQL-invoked routine *R* is active.
 - e) Indicate in the routine execution context of *RSC* whether or not containing SQL, reading SQL-data, or modifying SQL-data is permitted.
NOTE 153 – Such an indication is derived from the routine descriptor of *R*.
 - f) Set the SQL-session user identifier of *RSC* to the current user identifier of *CSC*.
 - g) Set the authorization stack of *RSC* to empty.
 - h) Append a new pair of identifiers to the authorization stack of *RSC* such that the user identifier is the SQL-session user identifier of *RSC* and the role name is the current role name of *CSC*.

i) Case:

i) If *R* is an external routine, then:

- 1) If the external security characteristic of *R* is DEFINER, then in that new pair of identifiers:
 - A) If the routine authorization identifier is a user identifier, then set the user identifier of *RSC* to the routine authorization identifier of *R* and set the role name of *RSC* to the null value.
 - B) Otherwise, set the role name of *RSC* to the routine authorization identifier of *R* and set the user identifier of *RSC* to the null value.
- 2) If the external security characteristic of *R* is IMPLEMENTATION DEFINED, then in that new pair of identifiers set the user identifier and the role name of *RSC* to implementation-defined values.
- 3) If the external security characteristic of *R* is INVOKER, then in that new pair of identifiers maintain the user identifier and the role name of *RSC* unchanged.
- 4) Set the current authorization identifier of *RSC* to be the external routine authorization identifier of *R*.
- 5) Set the routine SQL-path of *RSC* to be the external routine SQL-path of *R*.
- 6) If *R* possibly contains SQL, possibly reads SQL-data, or possibly modifies SQL-data, then set the SQL-session module of *RSC* to be the module *M* of *P*; otherwise, set the SQL-session module of *RSC* to be an implementation-defined module.

ii) Otherwise:

- 1) In that new pair of identifiers, set the user identifier of *RSC* to be the routine authorization identifier of *R* and set the role name of *RSC* to be the null value.
- 2) Set the routine SQL-path of *RSC* to be the routine SQL-path of *R*.
- 3) Set the SQL-session module of *RSC* to be the SQL-session module of *CSC*.

j) *RSC* becomes the current SQL-session context.

7) Case:

- a) If *R* possibly contains SQL and containing SQL is not permitted, then an exception condition is raised: *external routine exception — containing SQL not permitted*.
- b) If *R* possibly reads SQL-data and reading SQL-data is not permitted, then:
 - i) If *R* is an external routine, then an exception condition is raised: *external routine exception — reading SQL-data not permitted*.
 - ii) Otherwise, an exception condition is raised: *SQL routine exception — reading SQL-data not permitted*.

- c) If R possibly modifies SQL-data and modifying SQL-data is not permitted, then:
 - i) If R is an external routine, then an exception condition is raised: *external routine exception — modifying SQL-data not permitted*.
 - ii) Otherwise, an exception condition is raised: *SQL routine exception — modifying SQL-data not permitted*.
- d) If R does not possibly modify SQL-data, then the routine execution context is set to indicate that modifying SQL-data is not permitted.
- e) If R does not possibly read SQL-data, then the routine execution context is set to indicate that neither modifying nor reading SQL-data is permitted.
- f) If R does not possibly contain SQL, then the routine execution context is set to indicate that neither modifying SQL-data, reading SQL-data, nor containing SQL is permitted.

NOTE 154 – Otherwise, the routine execution context is unaltered.

8) If R is an SQL routine, then

Case:

- a) If R is an SQL-invoked method whose routine descriptor does not include a STATIC indication and if CP_1 is the null value, then:
 - i) Let RV be the null value.
 - ii) If R is a mutator, then an exception condition is raised: *data exception — null instance used in mutator function*.
- b) If R is a null-call function and if any of CPV_i is the null value, then let RV be the null value.
- c) 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-transaction statement or an SQL-connection statement, then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted*.
 - iv) If reading SQL-data is not permitted and, before the completion of the execution of the <SQL routine body> of R , an attempt is made to execute an <SQL procedure statement> that possibly reads SQL-data, then an exception condition is raised: *SQL routine exception — reading SQL-data not permitted*.
 - v) If modifying SQL-data is not permitted and, before the completion of the execution of the <SQL routine body> of R , an attempt is made to execute an SQL-data change statement or an SQL-schema statement, then an exception condition is raised: *SQL routine exception — modifying SQL-data not permitted*.

- vi) If there is an unhandled exception or completion condition other than *successful completion* at completion of the execution of the <SQL routine body> or *R*, then that condition is re-raised by the <routine invocation>.
 - vii) If *R* is an SQL-invoked function, then:
 - 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 155 — “Returned value” is defined in Subclause 15.2, “<return statement>”.
 - viii) 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*.
- 9) 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) For *i* ranging from 1 (one) to *PN*, the *i*-th entry in *ESPL* is set to *CPV_i*.
 - B) For *i* equal to *PN*+1, the *i*-th entry in *ESPL* is the *result data item*.
 - C) For *i* ranging from (*PN*+1)+1 to (*PN*+1)+*N*, the *i*-th entry in *ESPL* is the *SQL indicator argument* corresponding to *CPV_i*.
 - D) For *i* equal to (*PN*+1)+*N*+1, the *i*-th entry in *ESPL* is the SQL indicator argument corresponding to the result data item.
 - E) For *i* equal to (*PN*+1)+(*N*+1)+1, the *i*-th entry in *ESPL* is the *exception data item*.
 - F) For *i* equal to (*PN*+1)+(*N*+1)+2, the *i*-th entry in *ESPL* is the *routine name text item*.
 - G) For *i* equal to (*PN*+1)+(*N*+1)+3, the *i*-th entry in *ESPL* is the *specific name text item*.
 - H) For *i* equal to (*PN*+1)+(*N*+1)+4, the *i*-th entry in *ESPL* is the *message text item*.
 - I) If *R* is an array-returning external function, then for *i* equal to (*PN*+1)+(*N*+1)+5, the *i*-th entry in *ESPL* is the *save area data item* and for *i* equal to (*PN*+1)+(*N*+1)+6, the *i*-th entry in *ESPL* is the *call type data item*.
 - J) Set the value of the SQL indicator argument corresponding to the result data item (that is, SQL argument value list entry (*PN*+1)+*N*+1) to 0 (zero).

- K) For i ranging from 1 (one) to PN , if CPV_i is the null value, then set entry $(PN+1)+i$ (that is, the i -th SQL indicator argument corresponding to CPV_i to -1); otherwise, set entry $(PN+1)+i$ (that is, the i -th SQL indicator argument corresponding to CPV_i) to 0 (zero).
 - L) If R is an array-returning external function, then set the value of the save data item (that is, SQL argument value list entry $(PN+1)+(N+1)+5$) to 0 (zero) and set the value of the call type data item (that is, SQL argument value list entry $(PN+1)+(N+1)+6$) 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 .
 - 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) For i ranging from 1 (one) to PN , if any 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 , the i -th entry in $ESPL$ is set to CPV_i .
- 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 an SQL-invoked method whose routine descriptor does not include a STATIC indication and if CPV_1 is the null value, then P is assumed to have been executed. If, in addition, R is a mutator, then an exception condition is raised: *data exception — null instance used in mutator function*.
- ii) 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.

iii) Otherwise:

- 1) If R is not an array-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, MUMPS, PASCAL, or PLI, let the *operative data type correspondences table* be Table 18, “Data type correspondences for Ada”, Table 19, “Data type correspondences for C”, Table 20, “Data type correspondences for COBOL”, Table 21, “Data type correspondences for Fortran”, Table 22, “Data type correspondences for MUMPS”, Table 23, “Data type correspondences for Pascal”, or Table 24, “Data type correspondences for PLI”, 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 10.13, “Execution of array-returning functions”, are applied with AR , $ESPL$, and P as *ARRAY*, *EFFECTIVE SQL PARAMETER LIST*, and *PROGRAM*, respectively.
- 3) If, before the completion of any execution of P , an attempt is made to execute an SQL-transaction statement or an SQL-connection statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted*.
- 4) If containing SQL is not permitted and, before the completion of any execution of P , an attempt is made to execute an <SQL procedure statement> that possibly contains SQL, then an exception condition is raised: *external routine exception — containing SQL not permitted*.
- 5) If reading SQL-data is not permitted and, before the completion of any execution of P , an attempt is made to execute an <SQL procedure statement> that possibly reads SQL-data, then an exception condition is raised: *external routine exception — reading SQL-data not permitted*.

- 6) If modifying SQL-data is not permitted and, before the completion of any execution of *P*, an attempt is made to execute an SQL-data change statement or an SQL-schema statement, then an exception condition is raised: *external routine exception — modifying SQL-data not permitted*.
 - 7) If the language specifies ADA (respectively C, COBOL, FORTRAN, MUMPS, PASCAL, PLI) and *P* is not a standard-conforming Ada program (respectively C, COBOL, Fortran, MUMPS, 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) For every open cursor *CR* that is associated with *RSC* and that is defined by a <declare cursor> that is contained in the <SQL-client module definition> of *P*:
 - A) The following SQL-statement is effectively executed:
`CLOSE CR`
 - B) *CR* is destroyed.
 - 2) Every instance of created local temporary tables and every instance of declared local temporary tables that is associated with *RSC* is destroyed.
 - 3) 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*.

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 *external routine exception*, then an exception condition is raised: *external routine exception*, using a subclass code equal to the final three characters of the value of the exception data item.
 - 3) If the first two characters of the exception data item are equal to the SQLSTATE condition code class value for *warning* and the third character of the exception data item is 'H', 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.
 - 4) Otherwise, an exception condition is raised: *external routine invocation exception — invalid SQLSTATE returned*.
 - iii) If the exception data item is not '00000' and *R* specified PARAMETER STYLE SQL, then the message text item is stored in the 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, 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* 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 a null-call function, *R* is not an array-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 not a null-call function, *R* is an array-returning external function, and *R* specifies PARAMETER STYLE SQL, then let *ERDI* be *AR*.
- III) If *R* is not a null-call function and *R* specifies PARAMETER STYLE GENERAL, then let *ERDI* be the value returned from *P*.
NOTE 156 – 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 character large object value corresponding to *ERDI*.
- 3) If *RT* is array type, then let *RDI* be the array value corresponding to *ERDI*.
- 4) 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 case>; 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 character large object value 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 user-defined type, then let *RDI* be the user-defined type value corresponding to *ERDI*.

- 2) Otherwise, if *CRT* is a user-defined type, then:

- a) Let *TSF* be the SQL-invoked routine identified by the specific name of the to-sql function associated with the result of *R*.
- b) Case:
 - i) 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:
CRT()
and whose second element is *ERDI*, and the subject routine *TSF*.
 - ii) 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*.
- c) Let *RDI* be the result of invocation of *TSF*.

- 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*, $1 \text{ (one)} \leq i \leq PN$, 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 157 – In this case, either R specifies PARAMETER STYLE SQL and entry $(PN+1)+i$ in SQL (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 character large object value corresponding to EV_i .
- III) If T_i is an array type, then CPV_i is set to the array value corresponding to EV_i .
- IV) 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:

- i) 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:

$T_i()$

and whose second element is EV_i , and the subject routine TSF_i .

- ii) 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 .

10) Case:

- a) If R is an SQL-invoked function, then:
 - i) Let $ERDT$ be the effective returns data type of the <routine invocation>.
 - ii) Let the result of the <routine invocation> be the result of assigning RV to a target of declared type $ERDT$ according to the rules of Subclause 9.2, "Store assignment".
- 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>, then CPV_i is assigned to TS_i according to the rules of Subclause 9.1, "Retrieval assignment".
 - ii) If TS_i is the <SQL parameter name> of an SQL parameter of an SQL-invoked routine, then CPV_i is assigned to TS_i according to the rules of Subclause 9.2, "Store assignment".
- 11) If the subject routine is a procedure whose descriptor PR includes a maximum number of dynamic result sets that is greater than zero, then a sequence of result sets RRS is returned to INV .
- a) Let MAX be maximum number of dynamic result sets included in PR .
 - b) Let OPN be the actual number of result set cursors declared in the body of the subject routine that remain open when control is returned to INV .
 - 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) Let FRC be the ordered set of result set cursors that remain open when PR returns to INV . Let FRC_i , $1 \leq i \leq RTN$, be the i -th cursor in FRC , let $FRCN_i$ be the <cursor name> that identifies FRC_i , and let RS_i be the result set of FRC_i .
 - e) Let NXT_i , $1 \leq i \leq RTN$, be the ordinal number of the row of RS_i that would be retrieved if the following SQL-statement were executed:

FETCH NEXT FROM $FRCN_i$ INTO . . .
 - f) Let TOT_i , $1 \leq i \leq RTN$, be the original cardinality of RS_i when established by the opening of FRC_i .
 - g) Let RRS be the ordered set of returned result sets RRS_i , $1 \leq i \leq RTN$, comprising the rows of RS_i at ordinal positions $ROW_{i,j}$, $NXT_i \leq j \leq TOT_i$.
 - h) A completion condition is raised: *warning — dynamic result sets returned.*

- i) RS_i , $1 \leq i \leq RTN$, is returned to *INV*.
- 12) 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 transaction access mode in *CSC* to the value of the current transaction access mode in *RSC*.
 - c) Set the value of the current transaction isolation level in *CSC* to the value of the current transaction isolation level in *RSC*.
 - d) Set the value of the current transaction diagnostics area limit in *CSC* to the value of the current transaction diagnostics area limit in *RSC*.
 - e) Replace the identities of all instances of global temporary tables in *CSC* with the identities of the instances of global temporary tables in *RSC*.
- 13) *CSC* becomes the current SQL-session context.

Conformance Rules

- 1) Without Feature S023, “Basic structured types”, conforming SQL language shall not specify a <generalized expression>.
- 2) Without Feature S201, “SQL routines on arrays”, the declared type of an <SQL argument> shall not be an array type.

10.5 <privileges>

10.5 <privileges>

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 <translation name>
    | TYPE <user-defined type 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>, <translation name>, or <user-defined type name>, then <privileges> shall 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 <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> shall 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) Case:
 - a) If a <grantor> of CURRENT_USER is specified and the current user identifier is the null value, then an exception condition is raised: *invalid grantor*.
 - b) If a <grantor> of CURRENT_ROLE is specified and the current role name is the null value, then an exception condition is raised: *invalid grantor*.

10.5 <privileges>

- 2) A <grantee> of PUBLIC denotes at all times a list of <grantee>s containing all of the <authorization identifier>s in the SQL-environment.
- 3) The set of *applicable roles* for an <authorization identifier> consists of all roles defined by the role authorization descriptors whose grantee is that <authorization identifier> or PUBLIC, together with all other roles they contain.
- 4) The set of *user privileges* for a <user identifier> consists of all privileges defined by the privilege descriptors whose grantee is either that <user identifier> or PUBLIC.
- 5) The set of *role privileges* for a <role name> consists of all privileges defined by the privilege descriptors whose grantee is either that <role name>, PUBLIC, or one of the applicable roles of that <role name>.
- 6) The set of *applicable privileges* for an <authorization identifier> is defined to be:
 - a) If that <authorization identifier> is a <user identifier>, then the set of user privileges for that <authorization identifier>.
 - b) If that <authorization identifier> is a <role name>, then the set of role privileges for that <authorization identifier>.
- 7) The phrase *enabled roles* refers to:
 - a) If the value of the current role name of the current SQL-session is a null value, then the empty set.
 - b) Otherwise, the set of roles defined by the current role name of the current SQL-session together with its applicable roles.
- 8) The phrase *enabled authorization identifiers* refers to the set of <authorization identifier>s defined by the enabled roles together with the current user identifier of the current SQL-session, if its value is not a null value.
- 9) The phrase *enabled privileges* refers to:
 - a) If the value of the current role name of the current SQL-session is a null value, then the empty set.
 - b) Otherwise, the set of privileges defined by the role privileges of the current role name of the current SQL-session.
- 10) The phrase *current user privileges* refers to:
 - a) If the value of the current user identifier of the current SQL-session is a null value, then the empty set.
 - b) Otherwise, the set of privileges defined by the user privileges of the current user identifier of the current SQL-session.
- 11) The phrase *current privileges* refers to the set of privileges defined by the current user privileges together with those defined by the enabled privileges.
- 12) A role *A* identified by <role name> is said to contain the set of roles identified by role authorization descriptors as having been granted to *A*, together with all other roles that are contained by roles in the set.

- 13) SELECT (<column name>) specifies the SELECT privilege on the indicated column and implies one or more column privilege descriptors.
- 14) 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.
- 15) 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.
- 16) 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.
- 17) 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.
- 18) 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.
- 19) *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", an <action> shall not specify TRIGGER.
- 3) Without Feature S081, "Subtables", an <action> shall not specify UNDER on an <object name> that specifies a <table name>.
- 4) Without Feature S023, "Basic structured types", an <action> shall not specify UNDER on an <object name> that specifies a <user-defined type name> that identifies a structured type.
- 5) Without Feature S024, "Enhanced structured types", an <action> shall not specify USAGE on an <object name> that specifies a <user-defined type name> that identifies a structured type.
- 6) Without Feature T281, "SELECT privilege with column granularity", an <action> that specifies SELECT shall not contain a <privilege column list>.

10.5 <privileges>

- 7) Without Feature F731, "INSERT column privileges", an <action> that specifies INSERT shall not contain a <privilege column list>.
- 8) Without Feature F691, "Collation and translation", in conforming SQL language, an <object name> shall not specify COLLATION or TRANSLATION.
- 9) Without Feature F451, "Character set definition", or Feature F461, "Named character sets", in conforming SQL language, an <object name> shall not specify CHARACTER SET.
- 10) Without Feature F251, "Domain support", in conforming SQL language, an <object name> shall not specify DOMAIN.
- 11) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <privilege method list>.

10.6 <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 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, GRAPHIC_IRV, ASCII_GRAPHIC, LATIN1, ISO8BIT, ASCII_FULL, UNICODE, and ISO10646, with definitions as specified in Subclause 4.2.4, “Named character sets”.
- 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>. If the <character set specification> is not contained in a <schema definition>, then the schema identified by the explicit or implicit qualifier of the <character set name> shall include the descriptor of C. If the <character set specification> is contained in a <schema definition> S, then S shall include a <schema element> that creates 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 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.

NOTE 158 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

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, implied by the <standard character set name>, are defined by the standard defining the character set identified by that <standard character set name>. The default collating sequence 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, implied by the <implementation-defined character set name>, are implementation-defined. The default collating sequence of the character set and whether the collating sequence 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 159 – The default collating sequence of the character set is defined as in Subclause 11.30, “<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 F451, “Character set definition”, or Feature F461, “Named character sets”, conforming SQL language shall not contain a <character set specification>.

10.7 <specific routine designator>

Function

Specify an SQL-invoked routine.

Format

```

<specific routine designator> ::=
    SPECIFIC <routine type> <specific name>
    | <routine type> <member name> [ FOR user-defined type ]

<routine type> ::=
    ROUTINE
    | FUNCTION
    | PROCEDURE
    | [ INSTANCE | STATIC ] METHOD

<member name> ::= <schema qualified routine name> [ <data type list> ]

<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 neither INSTANCE nor STATIC is specified, then INSTANCE is implicit.
- 3) If a <member name> *MN* is specified, then:
 - a) If <user-defined type> is specified, then <routine type> shall specify METHOD. If METHOD is specified, then <user-defined type> shall be specified.
 - b) 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 function that is not an SQL-invoked method in the schema identified by *SN* whose <schema qualified routine name> is *RN* such that for all *i* the declared type of its *i*-th SQL parameter is identical to the *i*-th <data type> in the <data type list> of *MN*. 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 *SN* whose <schema qualified routine name> is *RN* such that for all *i* the declared type of its *i*-th SQL parameter is identical to the *i*-th <data type> in the <data type list> of *MN*. The <specific routine designator> identifies that SQL-invoked function.

10.7 <specific routine designator>

- 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 <user-defined type> such that for all i , the declared data type of its i -th SQL parameter is identical to the i -th <data type> in the <data type list> of MN . The <specific routine designator> identifies that static SQL-invoked method.
 - B) Otherwise, there shall be exactly one instance SQL-invoked method of the type identified by <user-defined type> such that for all i , the declared data type of its i -th SQL parameter in the unaugmented <SQL parameter declaration list> is identical to the i -th <data type> in the <data type list> of MN . The <specific routine designator> identifies that SQL-invoked method.
- 4) If <routine type> specifies ROUTINE, then there shall be exactly one SQL-invoked routine in the schema identified by SN whose <schema qualified routine name> is RN such that for all i the declared type of its i -th SQL parameter is identical to the i -th <data type> in the <data type list> of MN . 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 SN 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 SN 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 <user-defined type>. The <specific routine designator> identifies that static SQL-invoked method.
 - B) Otherwise, there shall be exactly one instance SQL-invoked method of the user-defined type identified by <user-defined type> that is not a static SQL-invoked method. The <specific routine designator> identifies that SQL-invoked method.
- 4) If <routine type> specifies ROUTINE, then there shall be exactly one SQL-invoked routine in the schema identified by SN 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 function that is not an SQL-invoked method. 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 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 any <specific routine designator> that specifies METHOD.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

10.8 <collate clause>

Function

Specify a default collating sequence.

Format

<collate clause> ::= COLLATE <collation name>

Syntax Rules

- 1) Let *C* be the <collation name> contained in the <collate clause>. If the <collate clause> is not contained in a <schema definition>, then the schema identified by the explicit or implicit qualifier of the <collation name> shall include the descriptor of *C*. If the <collate clause> is contained in a <schema definition> *S*, then the schema identified by the explicit or implicit qualifier of the <collation name> shall include the descriptor of *C* or *S* shall contain a <schema element> that creates the descriptor of *C*.

Access Rules

- 1) Case:
 - a) If <collate clause> is contained 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*.

NOTE 160 – “applicable privileges” and “current privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

None.

Conformance Rules

- 1) Without Feature F691, “Collation and translation”, conforming SQL language shall not contain any <collate clause>.

10.9 <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 a <constraint name 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 specified or implicit <schema name> of the containing <schema definition>.
- 2) The <qualified identifier> of <constraint name> shall not be equivalent to the <qualified identifier> of the <constraint name> of any other constraint defined in the same schema.
- 3) If <constraint check time> is not specified, then INITIALLY IMMEDIATE is implicit.
- 4) 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) A <constraint name> identifies a constraint. Let the identified constraint be *C*.
- 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*.

10.9 <constraint name definition> and <constraint characteristics>

- 4) If, on completion of any SQL-statement, the constraint mode of any constraint is immediate, then that constraint is effectively checked.

NOTE 161 – This includes the cases where SQL-statement is a <set constraints mode statement>, a <commit statement>, or the statement that causes a constraint with a constraint mode of *initially immediate* to be created.

- 5) When a constraint is effectively checked, if the constraint is not satisfied, then an exception condition is raised: *integrity constraint violation*. If this exception condition is raised as a result of executing a <commit statement>, then SQLSTATE is not set to *integrity constraint violation*, but is set to *transaction rollback — integrity constraint violation* (see the General Rules of Subclause 16.6, “<commit statement>”).

Conformance Rules

- 1) Without Feature F721, “Deferrable constraints”, conforming SQL language shall contain no explicit <constraint characteristics>.
NOTE 162 – This means that INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.
- 2) Without Feature F491, “Constraint management”, conforming SQL language shall contain no <constraint name definition>.

10.10 Execution of BEFORE triggers

Function

Defines 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 163 – Activation of triggers is defined in Subclause 4.35, “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 164 – Ordering of triggers is defined in Subclause 4.35, “Triggers”.

Access Rules

- 1) Let *TRN* be the trigger name of a trigger *TR*. Let *S* be the schema identified by the <schema name> explicitly or implicitly contained in *TRN*. The current authorization identifier during the execution of the <triggered SQL statement> of *TR* is the <authorization identifier> of the owner of *S*.

General Rules

- 1) For *k* ranging from 1 (one) to *NT*, apply the General Rules of Subclause 10.12, “Execution of triggers”, with *TR_k* as *TRIGGER* and *SC_k* as *STATE CHANGE*, respectively.

Conformance Rules

None.

10.11 Execution of AFTER triggers**10.11 Execution of AFTER triggers****Function**

Defines 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 165 – Activation of triggers is defined in Subclause 4.35, “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 166 – Ordering of triggers is defined in Subclause 4.35, “Triggers”.

Access Rules

- 1) Let *TRN* be the trigger name of a trigger *TR*. Let *S* be the schema identified by the <schema name> explicitly or implicitly contained in *TRN*. The current authorization identifier during the execution of the <triggered SQL statement> of *TR* is the <authorization identifier> of the owner of *S*.

General Rules

- 1) For k ranging from 1 (one) to *NT*, apply the General Rules of Subclause 10.12, “Execution of triggers”, with TR_k as *TRIGGER* and SC_k as *STATE CHANGE*, respectively.

Conformance Rules

None.

10.12 Execution of triggers

Function

Defines 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) Trigger *TR* is executed as follows.
Case:
 - a) If the triggered action *TA* included in the trigger descriptor of *TR* specifies **FOR EACH ROW**, then, for each row *R* in *SC* for which *TR* is not considered as executed, *TA* is invoked and *TR* is considered as executed for *R*.
 - b) If *TR* is not considered as executed for *SC*, then *TA* is invoked once and *TR* is considered as executed for *SC*.
- 3) When *TA* of *TR* is invoked,
Case:
 - a) If *TA* contains a <search condition> and the <search condition> is satisfied, then the <triggered SQL statement> of *TA* is executed.
 - b) If *TA* does not contain a <search condition>, then the <triggered SQL statement> of *TA* is executed.
- 4) When the <triggered SQL statement> *TSS* of *TA* is executed:
 - a) The <SQL procedure statement>s simply contained in *TSS* are effectively executed in the order in which they are specified in *TSS*.
 - b) If the execution of *TSS* is not successful, then an exception condition is raised: *trigger action exception*. The exception information associated with *TSS* is entered into the diagnostics area in a location other than the location corresponding to condition number 1 (one).

Conformance Rules

None.

10.13 Execution of array-returning functions

10.13 Execution of array-returning functions

Function

Defines 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*, $1 \text{ (one)} \leq i \leq EN$, be the *i*-th parameter in *ESPL*.
- 5) Let *E* be 0 (zero).
- 6) If the call type data item has a value of -1 (indicating "open call"), then *P* is executed with a list of *EN* parameters *P_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, MUMPS, PASCAL, or PLI, let the *operative data type correspondences* table be Table 18, "Data type correspondences for Ada", Table 19, "Data type correspondences for C", Table 20, "Data type correspondences for COBOL", Table 21, "Data type correspondences for Fortran", Table 22, "Data type correspondences for MUMPS", Table 23, "Data type correspondences for Pascal", or Table 24, "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*.
- 7) 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*).

10.13 Execution of array-returning functions

- b) If the exception data item is '02000' (corresponding to the completion condition *no data*):
- i) If entry $(PN+1)+N+1$ in *ESPL* (that is, SQL indicator argument $N+1$, corresponding to the result data item) 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*).
- 8) 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 P_i whose parameter names are PN_i and whose values are set as follows:
 - i) Depending on whether the language of *R* specifies ADA, C, COBOL, FORTRAN, MUMPS, PASCAL, or PLI, let the *operative data type correspondences* table be Table 18, "Data type correspondences for Ada", Table 19, "Data type correspondences for C", Table 20, "Data type correspondences for COBOL", Table 21, "Data type correspondences for Fortran", Table 22, "Data type correspondences for MUMPS", Table 23, "Data type correspondences for Pascal", or Table 24, "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.
 - ii) For i varying from 1 (one) to *EN*, the $\langle \text{data type} \rangle DT_i$ 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 .
 - iii) The value of PD_i is set to the value of ESP_i .
 - 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 entry $(PN+1)+N+1$ in *ESPL* (that is, SQL indicator argument $N+1$ corresponding to the result data item) is negative, then let the *E*-th element of *AR* be the null value.
 - B) Otherwise, let the *E*-th element of *AR* be the value of the result data item.
 - 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*).

10.13 Execution of array-returning functions

- 3) Otherwise, set the value of the call type data item to 1 (one) (indicating *close call*).
- 9) 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 P_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, MUMPS, PASCAL, or PLI, let the *operative data type correspondences table* be Table 18, "Data type correspondences for Ada", Table 19, "Data type correspondences for C", Table 20, "Data type correspondences for COBOL", Table 21, "Data type correspondences for Fortran", Table 22, "Data type correspondences for MUMPS", Table 23, "Data type correspondences for Pascal", or Table 24, "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 .

Conformance Rules

None.

10.14 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) If PM is a character type, then P shall be a character type and the length of PM shall be the length of P , and the character set of PM shall be the character set of P .
- 3) If PM is exact numeric, then P shall be exact numeric and the precision and scale of PM shall be the precision and scale of P .
- 4) If PM is approximate numeric, then P shall be approximate numeric and the precision of PM shall be the precision of P .
- 5) If PM is binary large object, then P shall be binary large object and the maximum length of PM shall be the maximum length of P .
- 6) If PM is bit string, then P shall be bit string and the length of PM shall be the length of P .
- 7) If PM is datetime data type, then P shall be datetime data type, P shall be with or without timezone according as to whether PM is with or without timezone, and the <time fractional seconds precision> of PM shall be the <time fractional seconds precision> of P .
- 8) If PM is INTERVAL, then P shall be INTERVAL and the <interval qualifier> of PM shall be the <interval qualifier> of P .
- 9) If PM is a collection type, then P shall be a collection type and the element type and the maximum cardinality (if any) of PM shall be the element type and the maximum cardinality (if any) of P .
- 10) If PM is a row type, then:
 - a) P shall be a row type.
 - b) The degree of PM shall be the degree of P .
 - c) Let N be the degree of PM .
 - d) The data type $DTFPM_i$, $1 \text{ (one)} \leq i \leq N$, of field FPM_i in PM and the data type $DTFP_i$ of field FP_i in P shall be compatible.
 - e) For i varying from 1 to N , the Syntax Rules of this Subclause are applied with $DTFPM_i$ and $DTFP_i$ the two data types.
- 11) If PM is a user-defined type, then P shall be a compatible user-defined type.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

10.15 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.

10.16 Determination of a from-sql function for an overriding method**10.16 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.

10.17 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.

10.18 Determination of a to-sql function for an overriding method**10.18 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.

11 Schema definition and manipulation

11.1 <schema definition>

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>
    | <translation definition>
    | <assertion definition>
    | <trigger definition>
    | <user-defined type definition>
    | <schema routine>
    | <grant statement>
    | <role definition>
    | <grant role statement>

```

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> appears in an <externally-invoked procedure> in an SQL-client module, then the effective <schema authorization identifier> and <schema name> during processing of the <schema definition> is the <schema authorization identifier> and <schema name> specified or implicit in the <schema definition>. 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 167 – <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> defines an SQL-schema *S* in a catalog.
- 2) The <schema authorization identifier> is the current authorization identifier for privilege determination for *S*.
- 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.
- 4) A schema descriptor *SDS* is created that describes *S*. *SDS* 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 of every component of *S*.

Conformance Rules

- 1) Without Feature T331, "Basic roles", conforming SQL language shall not contain any <role definition>.
- 2) Without Feature T331, "Basic roles", conforming SQL language shall not contain any <grant role statement>.
- 3) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain any <schema path specification>.
- 4) Without Feature F521, "Assertions", conforming SQL language shall not contain any <assertion definition>.
- 5) Without Feature F691, "Collation and translation", conforming SQL language shall not contain any <collation definition>.
- 6) Without Feature F691, "Collation and translation", conforming SQL language shall not contain any <translation definition>.
- 7) Without Feature F251, "Domain support", conforming SQL language shall not contain any <domain definition>.
- 8) Without Feature F451, "Character set definition", or Feature F461, "Named character sets", a <schema character set specification> shall not be specified.
- 9) Without Feature F451, "Character set definition", conforming SQL language shall not contain any <character set definition>.
- 10) Without Feature F171, "Multiple schemas per user", a <schema name clause> shall specify AUTHORIZATION and shall not specify a <schema name>.

11.2 <drop schema statement>

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, translations, triggers, user-defined types, SQL-invoked routines, or roles, and the <schema name> of *S* shall not be generally contained in the SQL routine body of any routine descriptor.

NOTE 168 – 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
```

- 5) Let *CD* be the <collation name> included in the descriptor of any collating sequence included in *S*. The following <drop collation statement> is effectively executed:

DROP COLLATION *CD* CASCADE

- 6) Let *TD* be the <translation name> included in the descriptor of any translation included in *S*. The following <drop translation 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 generally 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 *RO* be the name included in the descriptor of any role included in *S*. The following <drop role statement> is effectively executed:

DROP ROLE *RO* CASCADE

- 13) The descriptor of *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>

11.3 <table definition>

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>
    | OF <user-defined type>
      [ <subtable clause> ]
      [ <table element list> ]

<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>
    | <self-referencing column specification>
    | <column options>

<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>... ]
    [ <collate clause> ]

```


<subtable clause> ::=
 UNDER <supertable clause>

<supertable clause> ::= <supertable name>

<supertable name> ::= <table name>

<like clause> ::= LIKE <table name>

Syntax Rules

- 1) Let T be the table defined by the <table definition> TD .
- 2) If a <table definition> is contained in a <schema definition> SD and the <table name> contains a <local or schema qualifier>, then that <local or schema qualifier> shall be equivalent to the implicit or explicit <schema name> of SD .
- 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) 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 the <table name>.
- 5) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <table name>.
- 6) If LIKE is specified, then:
 - a) Let $T1$ be the table identified in the <like clause>.
 - b) Let nt be the number of columns in $T1$. Let C_i , $1 \text{ (one)} \leq i \leq nt$, be the columns of $T1$, in the order in which they appear in $T1$, let CN_i be the column name included in the column descriptor of C_i , and let DT_i be the data type included in the column descriptor of C_i . Let CD_1 be:

$CN_1 \quad DT_1$

If nt is greater than 1 (one), then let CD_i , $2 \leq i \leq nt$, be:

, $CN_i \quad DT_i$

The <like clause> is effectively replaced by CN_i , $1 \text{ (one)} \leq i \leq nt$.

NOTE 169 – <column constraint>s are not included in *columns*; <column constraint>s are effectively transformed to <table constraints>s and are thereby excluded.

- 7) If <subtable clause> is specified, then:
 - a) The <table name> contained in the <subtable clause> identifies a *direct supertable* of T , which shall be a base table. T is called a *direct subtable* of the direct supertable of T .
 - b) ST shall be a direct subtype of the structured type of the direct supertable of T .

11.3 <table definition>

- c) 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*.
 - d) The subtable family of *T* shall have exactly one maximal supertable.
 - e) The subtable family of *T* shall not include a member, other than *T* itself, whose associated structured type is *ST*.
 - f) *TD* shall not contain a <table constraint definition> that specifies PRIMARY KEY.
 - g) There shall exist some supertable of *T* whose table descriptor includes a unique constraint descriptor *UCD* such that the nullability characteristic included in the column descriptor whose column name is included in *UCD* is *known not nullable*.
 - h) 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* 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*.
 - i) *T* is a referenceable table.
- 8) Let the term *originally-defined column* of *T* refer to a column of *T* that corresponds to an originally-defined attribute of *ST*.
- 9) For every <column options> *CO*, <column name> shall be equivalent to the <column name> specified in some <column definition> *RCD* implicitly or explicitly contained in *TD* and shall not refer to an inherited column of *T*. Distinct <column options>s contained in *TD* shall specify distinct <column name>s.
- a) If *CO* specifies a <scope clause> *SC*, then let *CURITIBA* be the <column name> contained in *RCD* followed in turn by the <data type> or <domain name> contained in *RCD*, *SC*, the <default clause> (if any) contained in *RCD*, and every <column constraint definition> contained in *RCD*. *RCD* is replaced by *CURITIBA*.
 - b) If *CO* specifies <collate clause> *CC*, then let *CURITIBA* be the <column name> contained in *RCD* followed in turn by the <data type> or <domain name> contained in *RCD*, the <default clause> (if any) contained in *RCD*, every <column constraint definition> contained in *RCD*, and *CC*. *RCD* is replaced by *CURITIBA*.
 - c) If *CO* specifies <default clause> *DC*, then let *CURITIBA* be the <column name> contained in *RCD* followed in turn by the <data type> or <domain name> contained in *RCD*, *DC*, every <column constraint definition> contained in *RCD* and the <collate clause> (if any) contained in *RCD*. *RCD* is replaced by *CURITIBA*.
 - d) If *CO* specifies a non-empty list *CCDL* of <column constraint definition>s, then let *CURITIBA* be the <column name> contained in *RCD* followed in turn by the <data type> or <domain name> contained in *RCD*, the <default clause> (if any) contained in *RCD*, *CCDL* and the <collate clause> (if any) contained in *RCD*. *RCD* is replaced by *CURITIBA*.
- 10) If “OF <user-defined type>” is specified, then:
- a) The <user-defined type name> simply contained in <user-defined type> shall identify a structured type *ST*. Let the <table element list>, if specified, be *TEL1*.
 - b) *TEL1* shall not contain a <like clause>.

- c) If <subtable clause> is not specified, then <self-referencing column specification> shall be specified.
- d) “OF <user-defined type>” is effectively replaced by a <table element list> *TEL1 TEL2*, defined as follows.

TEL1 consists of *n* <table element>s, where *n* is the number of attribute descriptors included in the data type descriptor of *ST*. For each attribute descriptor *AD* included in the data type descriptor of *ST*, the corresponding <table element> in *TEL1* is the <column definition> *CN DT DC CC*, where:

- i) *CN* is the attribute name included in *AD*.
- ii) If *AD* includes a domain name *DN*, then *DT* is *DN*; otherwise, *DT* is some <data type> that, under the General Rules of Subclause 6.1, “<data type>”, would result in the creation of *AD*.
- iii) Case:
 - 1) If *AD* describes an inherited attribute *IA*, then *DC* is some <default clause> whose <default option> denotes the default value included in the column descriptor of the direct supercolumn of *IA*.
 - 2) Otherwise, *DC* is some <default clause> whose <default option> denotes the default value included in *AD*.
- iv) Case:
 - 1) If *AD* describes an inherited attribute *IA*, and the descriptor of the direct supercolumn of *IA* includes a <collation name> *COLIN*, then *CC* is “COLLATE *COLIN*”.
 - 2) If *AD* describes an inherited attribute *IA*, and the descriptor of the direct supercolumn of *IA* does not include a <collation name>, then *CC* is a zero-length string.
 - 3) If *AD* includes a <collation name> *COLIN*, then *CC* is “COLLATE *COLIN*”.
 - 4) Otherwise *CC* is a zero-length string.

If <table element list> *TEL* is specified and contains a <table element> that is not a <column definition>, then *TEL2* is a <comma> followed by those <table elements> of *TEL* that are not <column definition>s, the members of each adjacent pair being separated by a <comma>; otherwise *TEL2* is a zero-length string.

- 11) If ON COMMIT is specified, then TEMPORARY shall be specified.
- 12) If TEMPORARY is specified and ON COMMIT is not specified, then ON COMMIT DELETE ROWS is implicit.
- 13) If “OF <user-defined type>” is not specified, then <table element list> shall contain at least one <column definition>.
- 14) If <self-referencing column specification> is specified, then:
 - a) “OF <user-defined type>” shall be specified.
 - b) <subtable clause> shall not be specified.

11.3 <table definition>

- c) <table scope> shall not be specified.
- d) Let *RST* be the reference type REF(*ST*).

Case:

- i) If SYSTEM GENERATED is specified, then *RST* shall have a system-defined representation.
 - ii) If USER GENERATED is specified, then *RST* shall have a user-defined representation.
 - iii) If DERIVED is specified, then:
 - 1) *RST* shall have a derived representation.
 - 2) Let $A_1, A_2, \dots, A_n$ be the n attributes included in the list of attributes of the derived representation of *RST*.
 - 3) *TD* shall contain a <table constraint definition> that specifies a <unique constraint definition> whose <unique column list> contains the attribute names of $A_1, A_2, \dots, A_n$ in that order.
 - 4) For every attribute A_i , $1 \text{ (one)} \leq i \leq n$, *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.
- 15) Every referenceable table referenced by a <scope clause> contained in a <table element> 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.
- 16) If *TEL1* contains a <column options>, then *TD* shall specify OF <user-defined type name> and any <column definition> shall be contained before the first <column options>.
- 17) A <column option list> shall immediately contain either a <scope clause> or a <default clause> or at least one <column constraint definition> or a <collate clause>.
- 18) The scope of the <table name> is the <table definition>.

Access Rules

- 1) If a <table definition> is contained in an SQL-client module, then the enabled authorization identifiers shall include *A*.
- 2) If a <like clause> is contained in a <table definition>, then the applicable privileges of *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>.

NOTE 170 – “applicable privileges” are defined in Subclause 10.5, “<privileges>”.

- 4) If “OF <user-defined type name>” is specified, then the applicable privileges of *A* shall include USAGE on *ST*.

NOTE 171 – “applicable privileges” are defined in Subclause 10.5, “<privileges>”.

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) For each <column options *CO*>, if *CO* contains a <scope clause> *SC*, then let *CD* be the column descriptor identified by the <column name> specified in *CO*. The <table name> specified in *SC* is included in the reference type descriptor that is included in *CD*.
- 4) If <user-defined type> is specified, then:
 - a) Let *R* be the structured type identified by the <user-defined type name> simply contained in <user-defined type>.
 - b) *R* is the structured type associated with *T*.
- 5) 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*, included as follows:
 - i) If <self-referencing column specification> is specified, then a column descriptor in which:
 - 1) The name *CN* of the column is <self-referencing column name>.
 - 2) The data type descriptor is that generated by the <data type> “REF(*ST*) SCOPE(*TN*)”.
 - 3) The nullability characteristic is *known not nullable*.
 - 4) The ordinal position is 1 (one).
 - 5) The column is indicated to be self-referencing.
 - ii) The column descriptor of each inherited column of *T*. If one of these is the descriptor of an inherited self-referencing column, then the <data type> and scope included in that descriptor are replaced by *ST* and *TN*, respectively.
 - iii) The column descriptor of each originally-defined column of *T*.
 - c) If the table descriptor includes the column descriptor of a self-referencing column, then:
 - i) An indication that the table is a referenceable table.
 - ii) A table constraint descriptor that describes a unique constraint whose unique column is *CN*.

11.3 <table definition>

- iii) If SYSTEM GENERATED is specified, then an indication that the self-referencing column is a system-generated self-referencing column.
- iv) If DERIVED is specified, then an indication that the self-referencing column is a derived self-referencing column.
- v) If USER GENERATED is specified, 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>.
- e) If a <user-defined type> is specified, then the user-defined type name of *R*.
- f) The list (possibly empty) of the table names of each direct supertable of *T*, in the order in which they appear in <subtable clause>.
- g) An empty list indicating that *T* has no direct subtables.
- h) A non-empty set of functional dependencies, according to the rules given in Subclause 4.18, "Functional dependencies".
- i) A non-empty set of candidate keys.
- j) 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".
- k) 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.
- l) If TEMPORARY is specified, then
Case:
 - i) If ON COMMIT DELETE ROWS is specified, then the table descriptor includes an indication that ON COMMIT DELETE ROWS is specified.
 - ii) Otherwise, the table descriptor includes an indication that ON COMMIT PRESERVE ROWS is specified or implied.
- 6) In the descriptor of each direct supertable of *T*, *TN* is added to the end of the list of direct subtables.
- 7) 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".
- 8) 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 <user-defined type> is specified, then a table/method privilege descriptor is created on this table for every method of the structured type identified by the <user-defined type> 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”. If TEMPORARY is specified, then the grantee is “PUBLIC”; otherwise, the grantee is <authorization identifier> A.

- 9) 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*, and if the privilege is grantable let *WGO* be “WITH GRANT OPTION”; 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*

- 10) 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 contain one pair for each column of *T*, in the order of their ordinal position in *T*.

Conformance Rules

- 1) Without Feature T171, “LIKE clause in table definition”, a <table element> shall not be a <like clause>.
- 2) Without Feature F531, “Temporary tables”, conforming SQL language shall not specify TEMPORARY and shall not reference any global or local temporary table.
- 3) Without Feature S051, “Create table of type”, conforming SQL language shall not specify “OF <user-defined type>”.
- 4) Without Feature S043, “Enhanced reference types”, a <column option list> shall not contain a <scope clause>.
- 5) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not specify <self-referencing column specification>.
- 6) Without Feature S081, “Subtables”, conforming SQL language shall not specify <subtable clause>.

11.4 <column definition>

Function

Define a column of a base table.

Format

```
<column definition> ::=
    <column name>
    { <data type> | <domain name> }
    [ <reference scope check> ]
    [ <default clause> ]
    [ <column constraint definition>... ]
    [ <collate clause> ]

<column constraint definition> ::=
    [ <constraint name definition> ]
    <column constraint> [ <constraint characteristics> ]

<column constraint> ::=
    NOT NULL
    | <unique specification>
    | <references specification>
    | <check constraint definition>

<reference scope check> ::=
    REFERENCES ARE [ NOT ] CHECKED
    [ ON DELETE <reference scope check action> ]

<reference scope check action> ::=
    <referential action>
```

Syntax Rules

1) 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*.

- 2) Let *A* be the <authorization identifier> that owns *T*.
- 3) Let *C* be the <column name> of the <column definition>.
- 4) If <domain name> is specified, then let *D* be the domain identified by the <domain name>.
- 5) The declared type of the column is

Case:

- a) If <data type> is specified, then that data type.
 - b) Otherwise, the declared type of *D*.
- 6) If the declared type of the column is character string, then the collation of the column is

Case:

- a) If <collate clause> is specified, then the collation specified by that <collate clause>.
- b) If <domain name> is specified and *D* has a collation, then the collation of *D*.
- c) Otherwise, the default collation of the character set of the column.

NOTE 172 – The character set of a column is determined by its declared type.

- 7) 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.
- c) If *DT* is a <character string type> that identifies a character set that specifies a <collate clause> and the <column definition> does not contain a <collate clause>, then the <collate clause> of the <character string type> is implicit in the <column definition>.

- 8) If <collate clause> is specified, then *DT* shall specify a character string type.
- 9) If <data type> is a <reference type> that contains a <scope clause>, then <reference scope check> shall be specified; otherwise, <reference scope check> shall not be specified.
- 10) If REFERENCES ARE NOT CHECKED is specified, then <reference scope check action> shall not be specified.
- 11) If REFERENCES ARE CHECKED is specified and <reference scope check action> is not specified, then ON DELETE NO ACTION is implicit.
- 12) If <reference scope check> specifies REFERENCES ARE CHECKED, then let *RSCA* be the explicit or implicit <reference scope action>, let *STN* be the <table name> specified in the <scope clause> contained in <data type>, and let *SGCN* be the column name included in the column descriptor of the self-referencing column whose descriptor is included in the table descriptor identified by *STN*. The following <references specification> is implicit:

REFERENCES *STN* (*SGCN*) ON DELETE *RSCA*

- 13) If <data type> simply contains an <array specification>, a <row type>, or a <user-defined type name> that identifies a user-defined type descriptor whose degree is greater than 0 (zero), then let *CDTD* be the descriptor associated with that <array specification>, <row type>, or <user-defined type name>, respectively.

11.4 <column definition>

- 14) For every field descriptor *FD* generally included in *CTDD* that includes a data type descriptor *FDTD* that includes an indication that references are checked:
- Let *CNG* be the <column name> and let *TNG* be the <table name> immediately contained in the containing <table definition> or <alter table statement>.
 - Let *FN* be the <field name> included in *FD*.
 - Let *TND* be the <table name> included in the scope included in the data type descriptor included in *FDTD*.
 - Let *CND* be the column name included in the column descriptor of the self-referencing column whose descriptor is included in the table descriptor of the table identified by *TND*.
 - Let *QFN* be *TNG.CNG ... FN*, where “...” comprises a sequence of <identifier>s and <element reference>s separated by <period>s in which each <identifier> specifies a <field name> or an <attribute name>, such that the field identified by *FN* is properly referenced.
 - Let *FCN* be an implementation-dependent <constraint name> not equivalent to any <constraint name> in the schema containing the descriptor of the table identified by *TNG*.
 - The following <column constraint definition> is implicit:

```
CONSTRAINT FCN
CHECK ( QFN IN ( SELECT CND FROM TND ) )
```

- 15) For every attribute descriptor *AD* generally included in *CTDD* that includes a data type descriptor *ADTD* that includes an indication that references are checked:
- Let *CNG* be the <column name> and let *TNG* be the <table name> immediately contained in the containing <table definition> or <alter table statement>.
 - Let *AN* be the <attribute name> included in *AD*.
 - Let *TNL* be the <table name> included in the scope included in the data type descriptor included in *ADTD*.
 - Let *CND* be the column name included in the column descriptor of the self-referencing column whose descriptor is included in the table descriptor of the table identified by *TND*.
 - Let *QAN* be *TNG.CNG ... AN*, where “...” comprises a sequence of <identifier>s and <element reference>s separated by <period>s in which each <identifier> specifies a <field name> or an <attribute name>, such that the attribute identified by *AN* is properly referenced.
 - Let *ACN* be an implementation-dependent <constraint name> not equivalent to any <constraint name> in the schema containing the descriptor of the table identified by *TNG*.
 - The following <column constraint definition> is implicit:

```
CONSTRAINT ACN
CHECK ( QAN IN ( SELECT CND FROM TNL ) )
```

- 16) 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 173 – 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 174 – 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*.

- 17) If the <column definition> is not contained in a <schema definition>, then the schema identified by the explicit or implicit qualifier of the <domain name> shall include the descriptor of *D*. If the <column definition> is contained in a <schema definition> *S*, then *S* shall include a <schema element> that creates the descriptor of *D*.

Access Rules

- 1) If <domain name> is specified, then the applicable privileges of *A* shall include USAGE on *D*.
NOTE 175 – “applicable privileges” are defined in Subclause 10.5, “<privileges>”.
- 2) If a <data type> is specified that is one of the following:
- a) A user-defined type *U*.
 - b) A reference type whose referenced type is a user-defined type *U*.
 - c) An array type whose element type is a user-defined type *U*.
 - d) An array type whose element type is a reference type whose referenced type is a user-defined type *U*.
 - e) A row type with a subfield that has a declared type that is:
 - i) A user-defined type *U*.

11.4 <column definition>

- ii) A reference type whose referenced type is a user-defined type *U*.
- iii) An array type whose element type is a user-defined type *U*.
- iv) An array type whose element type is a reference type whose referenced type is a user-defined type *U*.

then the applicable privileges of *A* shall include USAGE on *U*.

NOTE 176 – “applicable privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) A <column definition> defines a column in a table.
 - 2) The <collate clause> specifies the default collating sequence for the column. If <collate clause> is not specified, then the default collating sequence is that used for comparisons of *Coercible* coercibility characteristic, as defined in Subclause 8.2, “<comparison predicate>”.
 - 3) If the <column definition> specifies <data type>, then a data type descriptor is created that describes the declared type of the column being defined.
 - 4) 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).
 - 5) 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>, then the data type descriptor of the declared type of the column.
 - ii) Otherwise, the domain of the column.
 - c) The ordinal position of the column, which is equal to the degree of *T*, unless the column is the self-referencing column of *T*, in which case it is 1 (one) and the ordinal position in every existing column descriptor in the table descriptor of *T* is increased by 1 (one).
 - d) The nullability characteristic of the column, determined according to the rules in Subclause 4.15, “Columns, fields, and attributes”.
- NOTE 177 – Both <column constraint definition>s and <table constraint definition>s must be analyzed to determine the nullability characteristics of all columns.
- e) If <default clause> is specified, then the <default option>.
 - f) If the <column definition> contains a <collate clause>, then the collation name. If <data type> is a reference type, then whether references are checked and whether <reference scope check action> is RESTRICT or SET NULL.

Conformance Rules

- 1) Without Feature F691, “Collation and translation”, conforming SQL language shall not contain any <collate clause>.
- 2) Without Feature F251, “Domain support”, a <column definition> shall not contain a <domain name>.
- 3) Without Feature F701, “Referential update actions”, a <column constraint> shall not contain an <update rule>.
- 4) Without Feature F191, “Referential delete actions”, a <column constraint> shall not contain a <delete rule>.
- 5) Without Feature F491, “Constraint management”, conforming SQL language shall not contain any <constraint name definition>.
- 6) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not specify REFERENCES ARE CHECKED.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

11.5 <default clause>

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_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>.
- 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 bit string, then the <literal> shall be a <bit string literal> or a <hex string literal>. If the length of the subject data type is fixed, then the length in bits of the <bit string literal> or <hex 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 bits of the <bit string literal> or <hex string literal> shall not be greater than the maximum length of the subject data type.
 - iii) If the subject data type is binary string, then the <literal> shall be a <binary string literal> that has an even number of <hexit>s. The length in octets of the <binary string literal> shall not be greater than the maximum length of the subject data type.
 - iv) 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.
 - v) If the subject data type is approximate numeric, then the <literal> shall be a <signed numeric literal>.
 - vi) If the subject data type is datetime, then the <literal> shall be a <datetime literal> and shall contain the same <primary datetime field>s as the subject data type.
 - vii) 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.
 - viii) If the subject data type is boolean, then the <literal> shall be a <boolean literal>.
 - ix) If the subject data type is a collection type, then the declared type of <literal> shall be that collection type.
- b) If CURRENT_USER, CURRENT_ROLE, SESSION_USER, or SYSTEM_USER 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 to the attribute descriptor, if the <default clause> is to apply to an attribute, or to the user-defined type descriptor, if the <default clause> is to apply to a user-defined type, is the <default option>.

- 2) If the subject data type is bit string with fixed length, the <default clause> specifies a <bit string literal>, and the length of the <bit string literal> is less than the fixed length of the column, then a completion condition is raised: *warning — implicit zero-bit padding*.

- 3) 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 bit string with variable length, then the value of the <literal>.
- v) If the subject data type is bit string with fixed length, then the value of the <literal> extended as necessary on the right with 0-valued bits to the length of the subject data type.
- vi) If the subject data type is binary string, then the value of the <literal>.
- vii) If the subject data type is datetime or interval, then the value of the <literal>.
- viii) If the subject data type is boolean, then the value of the <literal>.
- ix) If the subject data type is a collection type, then the value of the <literal>.
- b) If the <default option> specifies CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, 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, SESSION_USER, SYSTEM_USER, 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, SESSION_USER, CURRENT_PATH, or SYSTEM_USER 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.
- 4) The default value of a site is
- Case:
- a) If the data descriptor for the site includes a <default option>, then the value specified by that <default option>.
 - b) If the data descriptor for the site includes a <domain name> that identifies a domain descriptor that includes a <default option>, then the value specified by that <default option>.
 - c) 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 the default value of *CC* is obtained by applying the General Rules of this Subclause.
 - d) Otherwise, the null value.
- NOTE 178 – If <default option> specifies CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_ROLE 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.
- 5) 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*.

Conformance Rules

- 1) Without Feature S071, “SQL paths in function and type name resolution”, a <default option> shall not specify CURRENT_PATH.
- 2) Without Feature F321, “User authorization”, a <general value specification> shall not specify CURRENT_USER, SYSTEM_USER, or SESSION_USER.
NOTE 179 – Although CURRENT_USER and USER are semantically the same, in Core SQL, CURRENT_USER must be specified as USER.
- 3) Without Feature F321, “User authorization”, a <default option> shall not be CURRENT_USER, SESSION_USER, or SYSTEM_USER.
- 4) Without Feature T332, “Extended roles”, a <default option> shall not be CURRENT_ROLE.

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>.

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 the <constraint name> contained in the explicit or implicit <constraint name definition>.

The table constraint descriptor includes an indication of whether the constraint is deferrable or not deferrable and whether the initial constraint mode of the constraint is *deferred* or *immediate*.

Case:

- a) If <unique constraint definition> is specified, then the table constraint descriptor is a unique constraint descriptor that includes an indication of whether it was defined with PRIMARY KEY or UNIQUE, and the names of the unique columns specified in the <unique column list>.

- b) If <referential constraint definition> is specified, then the table constraint descriptor is a referential constraint descriptor that includes the names of the referencing columns specified in the <referencing columns> and the names of the referenced columns and referenced table specified in the <referenced table and columns>, the value of the <match type>, if specified, and the <referential triggered actions>, if specified.
- c) If <check constraint definition> is specified, then the table constraint descriptor is a table check constraint descriptor that includes the <search condition>.
- 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 table constraint is not satisfied if and only if

EXISTS (SELECT * FROM *T* WHERE NOT (*SC*))

is true.

Conformance Rules

- 1) Without Feature F491, “Constraint management”, conforming SQL language shall contain no <constraint name definition>.

11.7 <unique constraint definition>

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) The declared type of no column identified by any <column name> in the <unique column list> shall be based on a large object string type or an array type.
- 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  
        ( UCL ) IS NOT NULL
```

- 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.
NOTE 180 – Subclause 10.9, “<constraint name definition> and <constraint characteristics>”, specifies when a constraint is effectively checked.
- 2) The unique constraint is not satisfied if and only if

```
EXISTS ( SELECT * FROM TW WHERE NOT ( SC ) )
```

is true.

Conformance Rules

- 1) Without Feature F251, “Domain support”, conforming SQL language shall not specify UNIQUE(VALUE).
- 2) Without Feature T591, “UNIQUE constraints of possibly null columns”, if UNIQUE is specified, then the <column definition> for each column whose <column name> is contained in the <unique column list> shall specify NOT NULL.

11.8 <referential constraint definition>

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 the set of <column name>s contained in that <reference column list> shall be equal to the set of <column name>s contained in the <unique column list> of a unique constraint of the referenced table. 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) 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.
 - 9) If a <referential constraint definition> does not specify any <update rule>, then an <update rule> with a <referential action> of NO ACTION is implicit.
 - 10) If a <referential constraint definition> does not specify any <delete rule>, then a <delete rule> with a <referential action> of NO ACTION is implicit.

11.8 <referential constraint definition>

- 11) Let T be the referenced table. If the <referential constraint definition> is not contained in a <schema definition>, then the schema identified by the explicit or implicit qualifier of the <table name> shall include the descriptor of T . If the <referential constraint definition> is contained in a <schema definition> S , then S shall include a <schema element> that creates the descriptor of T .

Access Rules

- 1) The applicable privileges of the owner of T shall include REFERENCES for each referenced column.

NOTE 181 – “applicable privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) A <referential constraint definition> defines a referential constraint.
NOTE 182 – Subclause 10.9, “<constraint name definition> and <constraint characteristics>”, specifies when a constraint is effectively checked.
- 2) Let R_f be the referencing columns and let R_t be the referenced columns in the referenced table T . The referencing table and the referenced table satisfy the referential constraint if and only if:

Case:

- a) SIMPLE is specified or implicit and for each row of the referencing table, the <match predicate>

R_f MATCH SIMPLE (SELECT R_t FROM T)

is true.

- b) PARTIAL is specified and for each row of the referencing table, the <match predicate>

R_f MATCH PARTIAL (SELECT R_t FROM T)

is true.

- c) FULL is specified and for each row of the referencing table, the <match predicate>

R_f MATCH FULL (SELECT R_t FROM T)

is true.

- 3) 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 *unique matching row*.
- 4) 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.

- 5) 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 .
- 6) Let F be a subtable or supertable of the referencing table.
 - a) Let FL be the set of all columns of F . Let SRC be the set of referencing columns in F . Let SS be the set whose elements are the empty set and each subset of FL that contains at least one column in SRC . Let NSS be the number of sets in SS .
 - b) Let PMC be the set of referencing columns in F that correspond with the referenced columns. Let PSS be the set whose elements are the empty set and each subset of FL that contains at least one column in PMC . Let $PNSS$ be the number of sets in PSS .
 - c) Let UMC be the set of referencing columns that correspond with updated referenced columns. Let USS be the set whose elements are the empty set and each subset of FL that contains at least one column in UMC . Let $UNSS$ be the number of sets in USS .
- 7) If a row of the referenced table that has not previously been marked for deletion is marked for deletion, then

Case:

- a) If **SIMPLE** is specified or implicit, or if **FULL** is specified, then

Case:

- i) If the <delete rule> specifies **CASCADE**, then for every F :
 - 1) Every matching row in F is marked for deletion.
 - 2) If no SC_i has subject table F , trigger event **DELETE**, and an empty column list, then a new state change SC_j is added to SSC as follows:
 - A) The trigger event of SC_j is **DELETE**.
 - B) The subject table of SC_j is F .
 - C) The column list of SC_j is empty.

11.8 <referential constraint definition>

ii) If the <delete rule> specifies SET NULL, then:

- 1) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes *BTSS*:
 - A) For every *F*, for *k* ranging from 1 (one) to *NSS*, let *SS_k* be the *k*-th set of *SS*.
 - B) *BTSS* contains a state change *SC_k* as follows:
 - I) The trigger event of *SC_k* is UPDATE.
 - II) The subject table of *SC_k* is *F*.
 - III) The column list of *SC_k* is *SS_k*.
 - IV) The set of transitions of *SC_k* is a copy of the set of matching rows in *F*.
- 2) For every *F*, in every matching row in *F*, each referencing column in *F* is set to the null value.
- 3) For every *F*, for *k* ranging from 1 (one) to *NSS*, let *SS_k* be the *k*-th set of *SS*.

Case:

- A) If no *SC_i* has subject table *F*, trigger event UPDATE, and column list that is *SS_k*, then a new state change *SC_j* is added to *SSC* as follows:
 - I) The trigger event of *SC_j* is UPDATE.
 - II) The subject table of *SC_j* is *F*.
 - III) The column list of *SC_j* is *SS_k*.
 - IV) The set of transitions of *SC_j* is a copy of the set of matching rows in *F*.
- B) Otherwise, let *SC_j* be the state change in *SSC* that has subject table *F*, trigger event UPDATE, and a column list that is *SS_k*. A copy of the set of matching rows in *F* is added to the set of transitions of *SC_j*.

iii) If the <delete rule> specifies SET DEFAULT, then:

- 1) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes *BTSS*:
 - A) For every *F*, for *k* ranging from 1 (one) to *NSS*, let *SS_k* be the *k*-th set of *SS*.
 - B) *BTSS* contains a state change *SC_k* as follows:
 - I) The trigger event of *SC_k* is UPDATE.
 - II) The subject table of *SC_k* is *F*.
 - III) The column list of *SC_k* is *SS_k*.
 - IV) The set of transitions of *SC_k* is a copy of the set of matching rows in *F*.

11.8 <referential constraint definition>

- 2) For every F , in every unique matching row in F , each referencing column in F is set to the default value specified in the General Rules of Subclause 11.5, "<default clause>".
- 3) For every F , for k ranging from 1 (one) to NSS , let SS_k be the k -th set of SS .
Case:
 - A) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:
 - I) The trigger event of SC_j is UPDATE.
 - II) The subject table of SC_j is F .
 - III) The column list of SC_j is SS_k .
 - IV) The set of transitions of SC_j is a copy of the set of matching rows in F .
 - B) Otherwise, let SC_j be the state change in SSC that has subject table F , event UPDATE, and column list that is SS_k . A copy of the set of matching rows in F is added to the set of transitions of SC_j .
- 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 for every F :
 - 1) Every unique matching row in F marked for deletion.
 - 2) If no SC_i has subject table F , event DELETE, and an empty column list, then a new state change SC_j is added to SSC as follows:
 - A) The trigger event of SC_j is DELETE.
 - B) The subject table of SC_j is F .
 - C) The column list of SC_j is empty.
 - ii) If the <delete rule> specifies SET NULL, then:
 - 1) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes $BTSS$:
 - A) For every F , for k ranging from 1 (one) to NSS , let SS_k be the k -th set of SS .
 - B) $BTSS$ contains a state change SC_k as follows:
 - I) The trigger event of SC_k is UPDATE.
 - II) The subject table of SC_k is F .
 - III) The column list of SC_k is SS_k .
 - IV) The set of transitions of SC_k is a copy of the set of matching rows in F .

11.8 <referential constraint definition>

- 2) For every F , in every unique matching row in F , each referencing column is set to the null value.

- 3) For every F , for k ranging from 1 (one) to NSS , let SS_k be the k -th set of SS .

Case:

- A) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:
 - I) The trigger event of SC_j is UPDATE.
 - II) The subject table of SC_j is F .
 - III) The column list of SC_j is SS_k .
 - IV) The set of transitions of SC_j is a copy of the set of matching rows in F .
- B) Otherwise, let SC_j be the state change in SSC that has subject table F , trigger event UPDATE, and column list that is SS_k . A copy of the set of matching rows in F is added to the set of transitions of SC_j .

- iii) If the <delete rule> specifies SET DEFAULT, then:

- 1) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes $BTSS$:

- A) For every F , for k ranging from 1 (one) to NSS , let SS_k be the k -th set of SS .

- B) $BTSS$ contains a state change SC_k as follows:

- I) The trigger event of SC_k is UPDATE.
- II) The subject table of SC_k is F .
- III) The column list of SC_k is SS_k .
- IV) The set of transitions of SC_k is a copy of the set of matching rows in F .

- 2) For every F , in every unique matching row in F , each referencing column in F is set to the default value specified in the General Rules of Subclause 11.5, "<default clause>".

- 3) For every F , for k ranging from 1 (one) to NSS , let SS_k be the k -th set of SS .

Case:

- A) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:
 - I) The trigger event of SC_j is UPDATE.
 - II) The subject table of SC_j is F .
 - III) The column list of SC_j is SS_k .
 - IV) The set of transitions of SC_j is a copy of the set of matching rows in F .

11.8 <referential constraint definition>

- B) Otherwise, let SC_j be the state change in SSC that has subject table F , trigger event UPDATE, and a column list that is SS_k . A copy of the set of matching rows in F is added to the set of transitions of SC_j .
- 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 183 – Otherwise, the <referential action> is not performed.
- 8) If a non-null value of a referenced column in the referenced table is updated to a value that is distinct from the current value of that column, 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) The General Rules of Subclause 10.10, “Execution of BEFORE triggers”, are applied with the following new set of state changes $BTSS$:

A) For every F , for k ranging from 1 (one) to $PNSS$, let SS_k be the k -th set of PSS .

B) $BTSS$ contains a state change SC_k as follows:

I) The trigger event of SC_k is UPDATE.

II) The subject table of SC_k is F .

III) The column list of SC_k is SS_k .

IV) The set of transitions of SC_k is the set of matching rows in F .

- 2) For every F , in every matching row in F , each referencing column in F that corresponds with a referenced column is updated to the new value of that referenced column.

- 3) For every F , for k ranging from 1 (one) to $PNSS$, let SS_k be the k -th set of PSS .

Case:

- A) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:

I) The trigger event of SC_j is UPDATE.

II) The subject table of SC_j is F .

III) The column list of SC_j is SS_k .

IV) The set of transitions of SC_j is a copy of the set of matching rows in F .

- B) Otherwise, let SC_j be the state change in SSC that has subject table F , trigger event UPDATE, and a column list that is SS_k . A copy of the set of matching rows in F is added to the set of transitions of SC_j .

11.8 <referential constraint definition>

- ii) If the <update rule> specifies SET NULL, then

Case:

- 1) If SIMPLE is specified or implicit, then:

- A) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes *BTSS*:

- I) For every *F*, for *k* ranging from 1 (one) to *PNSS*, let *SS_k* be the *k*-th set of *PSS*.

- II) *BTSS* contains a state change *SC_k* as follows:

- 1) The trigger event of *SC_k* is UPDATE.
- 2) The subject table of *SC_k* is *F*.
- 3) The column list of *SC_k* is *SS_k*.
- 4) The set of transitions of *SC_k* is a copy of the set of matching rows in *F*.

- B) For every *F*, in every matching row in *F*, each referencing column in *F* that corresponds with a referenced column is set to the null value.

- C) For every *F*, for *k* ranging from 1 (one) to *NSS*, let *SS_k* be the *k*-th set of *SS*.

Case:

- I) If no *SC_i* has subject table *F*, trigger event UPDATE, and a column list that is *SS_k*, then a new state change *SC_j* is added to *SSC* as follows:

- 1) The trigger event of *SC_j* is UPDATE.
- 2) The subject table of *SC_j* is *F*.
- 3) The column list of *SC_j* is *SS_k*.
- 4) The set of transitions of *SC_j* is a copy of the set of matching rows in *F*.

- II) Otherwise, let *SC_j* be the state change in *SSC* that has subject table *F*, trigger event UPDATE, and a column list that is *SS_k*. A copy of the set of matching rows in *F* is added to the set of transitions of *SC_j*.

- 2) If <match type> specifies FULL, then:

- A) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes *BTSS*:

- I) For every *F*, for *k* ranging from 1 (one) to *NSS*, let *SS_k* be the *k*-th set of *SS*.

- II) *BTSS* contains a state change *SC_k* as follows:

- 1) The trigger event of *SC_k* is UPDATE.
- 2) The subject table of *SC_k* is *F*.
- 3) The column list of *SC_k* is *SS_k*.

11.8 <referential constraint definition>

- 4) The set of transitions of SC_k is a copy of the set of matching rows.
- B) For every F , in every matching row in F , each referencing column in F is set to the null value.
- C) For every F , for k ranging from 1 (one) to NSS , let SS_k be the k -th set of SS .
- Case:
- I) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:
- 1) The trigger event of SC_j is UPDATE.
 - 2) The subject table of SC_j is F .
 - 3) The column list of SC_j is SS_k .
 - 4) The set of transitions of SC_j is a copy of the set of matching rows in F .
- II) Otherwise, let SC_j be the state change in SSC that has subject table F , trigger event UPDATE, and a column list that is SS_k . A copy of the set of matching rows in F is added to the set of transitions of SC_j .
- iii) If the <update rule> specifies SET DEFAULT, then:
- 1) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes $BTSS$:
 - A) For every F , for k ranging from 1 (one) to $PNSS$, let SS_k be the k -th set of PSS .
 - B) $BTSS$ contains a state change SC_k as follows:
 - I) The trigger event of SC_k is UPDATE.
 - II) The subject table of SC_k is F .
 - III) The column list of SC_k is SS_k .
 - IV) The set of transitions of SC_k is the set of matching rows in F .
 - 2) For every F , in every matching row in F , the referencing column in F that corresponds with the referenced column is set to the default value specified in the General Rules of Subclause 11.5, "<default clause>".
 - 3) For every F , for k ranging from 1 (one) to $PNSS$, let SS_k be the k -th set of PSS .
- Case:
- A) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:
- I) The trigger event of SC_j is UPDATE.
 - II) The subject table of SC_j is F .
 - III) The column list of SC_j is SS_k .
 - IV) The set of transitions of SC_j is a copy of the set of matching rows in F .

11.8 <referential constraint definition>

- B) Otherwise, let SC_j be the state change in SSC that has subject table F , event UPDATE, and a column list that is SS_k . A copy of the set of matching rows is added to the set of transitions of SC_j .
- 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) The General Rules of Subclause 10.10, “Execution of BEFORE triggers”, are applied with the following new set of state changes $BTSS$:
 - A) For every F , for k ranging from 1 (one) to $UNSS$, let SS_k be the k -th set of USS .
 - B) $BTSS$ contains a state change SC_k as follows:
 - I) The trigger event of SC_k is UPDATE.
 - II) The subject table of SC_k is F .
 - III) The column list of SC_k is SS_k .
 - IV) The set of transitions of SC_k is a copy of the set of matching rows in F .
 - 2) For every F , for each unique matching row in F that contains a non-null value in the referencing column $C1$ in F that corresponds with the updated referenced column $C2$, $C1$ is updated to the new value V of $C2$, provided that, in all updated rows in the referenced table that formerly had, in the same 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 . Otherwise, an exception condition is raised: *triggered data change violation*.

NOTE 184 – 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.
 - 3) For every F , for k ranging from 1 (one) to $UNSS$, let SS_k be the k -th set of USS .

Case:

 - A) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:
 - I) The trigger event of SC_j is UPDATE.
 - II) The subject table of SC_j is F .
 - III) The column list of SC_j is SS_k .
 - IV) The set of transitions of SC_j is a copy of the set of matching rows in F .
 - B) Otherwise, let SC_j be the state change in SSC that has subject table F , trigger event UPDATE, and a column list that is SS_k . A copy of the set of matching rows is added to the set of affected rows of SC_j .

ii) If the <update rule> specifies SET NULL, then:

- 1) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes *BTSS*:
 - A) For every *F*, for *k* ranging from 1 (one) to *UNSS*, let *SS_k* be the *k*-th set of *USS*.
 - B) *BTSS* contains a state change *SC_k* as follows:
 - I) The trigger event of *SC_k* is UPDATE.
 - II) The subject table of *SC_k* is *F*.
 - III) The column list of *SC_k* is *SS_k*.
 - IV) The set of transitions of *SC_k* is a copy of the set of matching rows in *F*.
- 2) For every *F*, in every unique matching row in *F* that contains a non-null value in a referencing column in *F* that corresponds with the updated column, that referencing column is set to the null value.
- 3) For every *F*, for *k* ranging from 1 (one) to *NSS*, let *SS_k* be the *k*-th set of *SS*.

Case:

- A) If no *SC_i* has subject table *F*, trigger event UPDATE, and a column list that is *SS_k*, then a new state change *SC_j* is added to *SSC* as follows:
 - I) The trigger event of *SC_j* is UPDATE.
 - II) The subject table of *SC_j* is *F*.
 - III) The column list of *SC_j* is *SS_k*.
 - IV) The set of transitions of *SC_j* is a copy of the set of matching rows in *F*.
- B) Otherwise, let *SC_j* be the state change in *SSC* that has subject table *F*, trigger event UPDATE, and a column list that is *SS_k*. A copy of the set of matching rows is added to the set of transitions of *SC_j*.

iii) If the <update rule> specifies SET DEFAULT, then:

- 1) The General Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes *BTSS*:
 - A) For every *F*, for *k* ranging from 1 (one) to *UNSS*, let *SS_k* be the *k*-th set of *USS*.
 - B) *BTSS* contains a state change *SC_k* as follows:
 - I) The trigger event of *SC_k* is UPDATE.
 - II) The subject table of *SC_k* is *F*.
 - III) The column list of *SC_k* is *SS_k*.
 - IV) The set of transitions of *SC_k* is a copy of the set of matching rows in *F*.

11.8 <referential constraint definition>

- 2) For every F , in every unique matching row in F that contains a non-null value in the referencing column in F that corresponds with the updated column, that referencing column is set to the default value specified in the General Rules of Subclause 11.5, "<default clause>".

- 3) For every F , for k ranging from 1 (one) to $UNSS$, let SS_k be the k -th set of USS .

Case:

- A) If no SC_i has subject table F , trigger event UPDATE, and a column list that is SS_k , then a new state change SC_j is added to SSC as follows:

I) The trigger event of SC_j is UPDATE.

II) The subject table of SC_j is F .

III) The column list of SC_j is SS_k .

IV) The set of transitions of SC_j is a copy of the set of matching rows in F .

- B) Otherwise, let SC_j be the state change in SSC that has subject table F , trigger event UPDATE, and a column list that is SS_k . A copy of the set of matching rows is added to the set of transitions of SC_j .

- 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 185 — Otherwise, the <referential action> is not performed.

- 9) If any attempt is made within an SQL-statement to update some site to a value that is distinct from the value to which that site was previously updated within the same SQL-statement, then an exception condition is raised: *triggered data change violation*.
- 10) If a site in an object row is an <object column> of an <update statement: positioned> or <update statement: searched>, and there is any attempt within the same SQL-statement to delete the row containing that site, then an exception condition is raised: *triggered data change violation*.
- 11) If an <update rule> attempts to update a row that has been deleted by any <delete statement: positioned> that identifies some cursor CR that is still open or updated by any <update statement: positioned> that identifies some cursor CR that is still open or if a <delete rule> attempts to mark for deletion such a row, then a completion condition is raised: *warning — cursor operation conflict*.
- 12) For every row RMD that is marked for deletion, every subrow of RMD and every superrow of RMD is marked for deletion.
- 13) The general Rules of Subclause 10.10, "Execution of BEFORE triggers", are applied with the following new set of state changes $BTSS$.

For each table F_i that contains a row that is marked for deletion, $BTSS$ contains a state change SC_i as follows:

- a) The trigger event of SC_i is DELETE.
- b) The subject table of SC_i is F_i .
- c) The column list of SC_i is empty.

- d) The set of transitions of SC_i is a copy of the set of rows in F_i that are marked for deletion.
- 14) For each table F_i that contains a row that is marked for deletion, let SC_j be the state change in SSC that has subject table F_i , trigger event DELETE, and an empty column list. A copy of the rows in F_i that are marked for deletion is added to the set of affected rows of SC_j .
- 15) All rows that are marked for deletion are effectively deleted at the end of the SQL-statement, prior to the checking of any integrity constraints.

Conformance Rules

- 1) Without Feature T191, "Referential action RESTRICT", a <referential action> shall not be RESTRICT.
- 2) Without Feature F741, "Referential MATCH types", a <references specification> shall not specify MATCH.
- 3) Without Feature F191, "Referential delete actions", a <referential triggered action> shall not contain a <delete rule>.
- 4) Without Feature F701, "Referential update actions", a <referential triggered action> shall not contain an <update rule>.
- 5) Without Feature T201, "Comparable data types for referential constraints", the data type of each referencing column shall be the same as the data type of the corresponding referenced column.

11.9 <check constraint definition>

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 <target specification>.
- 2) The <search condition> shall not contain a <set function specification> that is not contained in a <subquery>.
- 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 no <subquery> in the <search condition> shall reference a temporary table defined by a <table definition> or a <temporary table declaration> that specifies ON COMMIT DELETE ROWS.
 - 5) The <search condition> shall not generally contain a <datetime value function> or a <value specification> that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, or CURRENT_PATH.
 - 6) The <search condition> shall not generally contain a <query specification> or a <query expression> that is possibly non-deterministic.
 - 7) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.

- 8) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.
- 9) Let *A* be the <authorization identifier> that owns *T*.

Access Rules

- 1) Let *TN* be any <table name> referenced in the <search condition>.

Case:

- a) If a <column name> is contained in the <search condition>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include REFERENCES for each <column name> of the table identified by *TN* contained in the <search condition>.
- b) Otherwise, the applicable privileges of the <authorization identifier> that owns the containing schema shall include REFERENCES for at least one column of the table identified by *TN*.

NOTE 186 – “applicable privileges” are defined in Subclause 10.5, “<privileges>”.

General Rules

- 1) A <check constraint definition> defines a check constraint.
NOTE 187 – Subclause 10.9, “<constraint name definition> and <constraint characteristics>”, specifies when a constraint is effectively checked. The General Rules that control the evaluation of a check constraint can be found in either Subclause 11.6, “<table constraint definition>”, or Subclause 11.23, “<domain definition>”, depending on whether it forms part of a table constraint or a domain 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*.

Conformance Rules

- 1) Without Feature F671, “Subqueries in CHECK constraints”, the <search condition> contained in a <check constraint definition> shall not contain a <subquery>.

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) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <table name>.
- 3) The schema identified by the explicit or implicit schema name of the <table name> shall include the descriptor of T .
- 4) The scope of the <table name> is the entire <alter table statement>.
- 5) T shall be a base table.
- 6) 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

- 1) Without Feature F033, “ALTER TABLE statement: DROP COLUMN clause”, conforming SQL language shall not specify <drop column definition>.
- 2) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not specify <alter column definition>.
- 3) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not specify <add table constraint definition>.
- 4) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not specify <drop table constraint definition>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

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.

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 . Every value in C is the default value for C .
NOTE 188 – The default value of a column is defined in Subclause 11.5, “<default clause>”.
NOTE 189 – 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 7.11, “<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>
```

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 an originally-defined column of *T*.
- 4) If <add column scope clause> or <drop column scope clause> is specified, then *C* shall not be the self-referencing column of *T*.

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>**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 *CO* 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) *RTD* shall not include a scope.
- 3) Let *UDTN* be the name of the referenced type included in *RTD*.
- 4) 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*.

Conformance Rules

- 1) Without Feature F381, “Extended schema manipulation”, and Feature S043, “Enhanced reference types”, conforming SQL language shall not contain any <add column scope clause>.

11.16 <drop column scope clause>

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 *CO* be the column identified by the <column name> in the containing <alter column definition>. The declared type of *C* shall be some reference type.
- 2) An *impacted dereference operation* is a <dereference operation> whose <value expression primary> is a column reference that identifies *C*, a <method reference> whose <value expression primary> is a column reference that identifies *C*, or a <reference resolution> whose <reference value expression> is a column reference that identifies *C*.
- 3) 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 or assertion descriptor.
 - d) The triggered action of any trigger descriptor.

NOTE 190 – 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 a <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
```

11.16 <drop column scope clause>

- 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*

- 4) For every table check constraint *C* whose table check constraint descriptor includes a <search condition> that contains an impacted dereference operation, let *CN* be the <constraint name> of *C* and let *TN* be the <table name> of the table whose descriptor includes descriptor of *C*. The following <alter table statement> is effectively executed for every *C* 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 *C* is made empty.

Conformance Rules

- 1) Without Feature F381, “Extended schema manipulation”, and Feature S043, “Enhanced reference types”, conforming SQL language shall not contain any <drop column scope clause>.

11.17 <drop column definition>

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.

NOTE 191 – 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), or REFERENCES without a <reference column list> in its <referenced table and columns>.

NOTE 192 – 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.

NOTE 193 – CN may be contained in an implicit trigger column list of a trigger descriptor.

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  $TR$ 
```

11.17 <drop column definition>

- 2) If *T* is the subject table of a trigger descriptor *TD* that contains an UPDATE trigger event with an implicit trigger column list, then *C* is removed from the trigger column list of *TD*.
- 3) 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

- 4) Let *R* be any SQL-invoked routine whose routine descriptor contains the <column name> of *C* in the <SQL routine body>. 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

- 5) If the column is not based on a domain, then its data type descriptor is destroyed.
- 6) The data associated with *C* is destroyed.
- 7) The descriptor of *C* is removed from the descriptor of *T*.
- 8) The descriptor of *C* is destroyed.
- 9) 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 specify <drop column definition>.

11.18 <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 194 – The nullability characteristic of a column is defined in Subclause 4.15, “Columns, fields, and attributes”.

Conformance Rules

- 1) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain an <add table constraint definition>.

11.19 <drop table constraint definition>**11.19 <drop table constraint definition>****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 V is a view that contains a <query specification> QS 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 table constraint TC is needed to conclude that $G \mapsto C$ is a known functional dependency in QS , then V is said to be *dependent on* TC .
- 5) 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 .
- 6) 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 .

NOTE 195 – If CASCADE is specified, then any such dependent object will be dropped by the effective execution of the <alter table statement> specified in 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) The descriptor of *TC* is removed from the descriptor of *T*.
- 5) 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 196 – The nullability characteristic of a column is defined in Subclause 4.15, “Columns, fields, and attributes”.

- 6) 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>.

11.20 <drop table statement>

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) If RESTRICT is specified, then T shall not have any subtables.
 - 6) If RESTRICT is specified, then T shall not be referenced in any of the following:
 - a) The <query expression> of any view descriptor.
 - b) The <search condition> of any table check constraint descriptor of any table other than T or the <search condition> of a constraint descriptor of an assertion descriptor.
 - c) The table descriptor of the referenced table of any referential constraint descriptor of any table other than T .
 - d) The scope of the declared type of a column of a table other than T and of the declared type of an SQL parameter of any SQL-invoked routine.
 - e) The <SQL routine body> of any SQL-invoked routine descriptor.
 - f) The scope of the declared type of an SQL parameter of any SQL-invoked routine.
 - g) The trigger action of any trigger descriptor.
- NOTE 197 – 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.
- 7) Let A be the <authorization identifier> that owns the schema identified by the <schema name> of the table identified by TN .
 - 8) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <table name>.

Access Rules

- 1) The enabled authorization identifiers shall include A .

General Rules

- 1) Let *ST* be the <table name> of any subtable of *T*. The following <drop table statement> is effectively executed without further Access Rule checking:

DROP TABLE *ST* CASCADE

- 2) If *T* is a referenceable table, then:

- a) Let *ST* be structured type associated with *T*.
- b) Let *RST* be the reference type whose referenced type is *ST*.
- c) Let *DT* be any table whose table descriptor includes a column descriptor that generally includes a field descriptor, an attribute descriptor, or an array descriptor that includes a reference type descriptor *RST* whose scope includes *TN*.

NOTE 198 – A descriptor that “generally includes” another descriptor is defined in Subclause 6.2.4, “Descriptors”, in ISO/IEC 9075-1.

- d) Let *DTN* be the name of the table *DT*.

- e) Case:

- i) If *DT* is a base table, then the following <drop table statement> is effectively executed without further Access Rule checking:

DROP TABLE *DTN* CASCADE

- ii) Otherwise, the following <drop view statement> is effectively executed without further Access Rule checking:

DROP VIEW *DTN* CASCADE

- 3) For every supertable of *T*, every superrow and every subrow of every row of *T* is effectively deleted at the end of the SQL-statement, prior to the checking of any integrity constraints.

NOTE 199 – This deletion creates neither a new trigger execution context nor the definition of a new state change in the current trigger execution context.

- 4) 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

- 5) Let *R* be any SQL-invoked routine whose routine descriptor contains the <table name> of *T* 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

- 6) 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*.

- 7) The descriptor of *T* is destroyed.

Conformance Rules

- 1) Without Feature F032, “CASCADE drop behavior”, a <drop behavior> of CASCADE shall not be specified in <drop table statement>.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 9075-2:1999

11.21 <view definition>

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 <user-defined type>
    [ <subview clause> ]
    [ <view element list> ]

<subview clause> ::= UNDER <table name>

<view element list> ::=
    <left paren>
        [ <self-referencing column specification> <comma> ]
        <view element> [ { <comma> <view element> }... ]
    <right paren>

<view element> ::= <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 have an element type that is a row type.
- 2) The <query expression> shall not contain a <target specification>.
- 3) 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>.
- 4) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <table name>.

11.21 <view definition>

- 5) 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>.
- 6) No <table reference> generally contained in the <query expression> shall identify any declared local temporary table.
- 7) 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*.
- 8) 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>)
```

- 9) The viewed table is updatable if and only if the <query expression> is updatable.
- 10) The viewed table is insertable-into if and only if the <query expression> is insertable-into.
- 11) If the <query expression> is a <query specification> that contains a <group by clause> or a <having clause> that is not contained in a <subquery>, then the viewed table defined by the <view definition> is a *grouped view*.
- 12) 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.
- 13) Equivalent <column name>s shall not be specified more than once in the <view column list>.
- 14) 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>.
- 15) No column in the table specified by <query expression> shall have a coercibility characteristic of *No collating sequence*.
NOTE 200 – The coercibility characteristic is described in Subclause 4.2.3, “Rules determining collating sequence usage”.
- 16) If WITH CHECK OPTION is specified, then the viewed table shall be updatable.
- 17) If WITH CHECK OPTION is specified and <levels clause> is not specified, then a <levels clause> of CASCADED is implicit.
- 18) 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*.

- 19) 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>.
- 20) <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 201 – It follows that the subviews of the supervIEWS of V together constitute the subtable family of V , every implication of which applies.

- 21) 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 <user-defined type> 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 <referenceable 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 , $1 \text{ (one)} \leq i \leq n$ be the attributes of ST .
- h) Let RT be the row type of QE .
- i) If <referenceable column specification> is specified, then:
 - i) <subview clause> shall not be specified.
 - ii) SYSTEM GENERATED shall not be specified.
 - iii) Let RST be the reference type $REF(ST)$.

Case:

- 1) If USER GENERATED is specified, then:

- A) RST shall have a user-defined representation.