



**International
Standard**

ISO/IEC 39075

Information technology — Database languages — GQL

*Technologies de l'information — Langages de base de
données — GQL*

**First edition
2024-04**

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword.....	xiv
Introduction.....	xv
1 Scope.....	1
2 Normative references.....	2
3 Terms and definitions.....	3
3.1 General terms and definitions.....	3
3.2 GQL-environment terms and definitions.....	4
3.3 GQL-catalog terms and definitions.....	6
3.4 Graph terms and definitions.....	7
3.5 Procedure and command terms and definitions.....	9
3.6 General syntax terms and definitions.....	12
3.7 Graph pattern terms and definitions.....	14
3.8 Value terms and definitions.....	15
3.9 Type terms and definitions.....	17
3.10 Temporal terms and definitions.....	19
4 Concepts.....	21
4.1 Use of terms.....	21
4.2 GQL-environments and their components.....	21
4.2.1 General description of GQL-environments.....	21
4.2.2 GQL-agents.....	22
4.2.3 GQL-implementations.....	22
4.2.3.1 Introduction to GQL-implementations.....	22
4.2.3.2 GQL-clients.....	23
4.2.3.3 GQL-servers.....	23
4.2.4 Basic security model.....	24
4.2.4.1 Principals.....	24
4.2.4.2 Authorization identifiers and privileges.....	24
4.2.5 GQL-catalog.....	24
4.2.5.1 General description of the GQL-catalog.....	24
4.2.5.2 GQL-directories.....	25
4.2.5.3 GQL-schemas.....	26
4.2.6 GQL-data.....	27
4.3 GQL-objects.....	27
4.3.1 General introduction to GQL-objects.....	27
4.3.2 References to GQL-schemas and GQL-objects.....	27
4.3.3 Primary objects and secondary objects.....	28
4.3.4 Properties and supported property value types.....	28
4.3.5 Graphs.....	29

4.3.5.1	Introduction to graphs.	29
4.3.5.2	Graph descriptors.	30
4.3.6	Binding tables.	30
4.4	Values.	32
4.4.1	General information about values.	32
4.4.2	Comparable values.	32
4.4.3	Properties of distinct values.	32
4.4.4	Reference values.	33
4.4.5	Material values and the null value.	33
4.5	GQL-sessions.	33
4.5.1	General description of GQL-sessions.	33
4.5.2	Session contexts.	34
4.5.2.1	Introduction to session contexts.	34
4.5.2.2	Session context creation.	35
4.5.2.3	Session context modification.	35
4.6	GQL-transactions.	35
4.6.1	General description of GQL-transactions.	35
4.6.2	Transaction demarcation.	36
4.6.3	Transaction isolation.	37
4.6.4	Encompassing transaction belonging to an external agent.	37
4.7	GQL-requests and GQL-programs.	38
4.7.1	General description of GQL-requests and GQL-programs.	38
4.7.2	GQL-request contexts.	38
4.7.2.1	Introduction to GQL-request contexts.	38
4.7.2.2	GQL-request context creation.	38
4.7.2.3	GQL-request context modification.	39
4.7.3	Execution of GQL-requests.	39
4.7.4	Working schema references.	40
4.7.5	Working graph site.	41
4.7.6	Execution stack.	41
4.7.7	Operations.	41
4.8	Execution contexts.	42
4.8.1	General description of execution contexts.	42
4.8.2	Execution context creation and initialization.	44
4.8.3	Execution context modification.	45
4.8.4	Execution outcomes.	45
4.9	Diagnostic information.	46
4.9.1	Introduction to diagnostic information.	46
4.9.2	GQL-status objects.	46
4.9.3	Conditions.	47
4.10	Procedures and commands.	49
4.10.1	General description of procedures and commands.	49
4.10.2	Procedures.	49
4.10.2.1	General description of procedures.	49
4.10.2.2	Named procedure descriptors.	50
4.10.2.3	Procedure execution.	50
4.10.2.4	Procedures classified by kind of side effects.	50

4.10.3	Commands.	51
4.10.4	GQL-procedures.	51
4.10.4.1	Introduction to GQL-procedures.	51
4.10.4.2	Binding variables and general parameters.	51
4.10.4.3	Statements.	52
4.10.4.4	Statements classified by use of working graph sites.	52
4.10.4.5	Statements classified by function.	52
4.11	Graph pattern matching.	53
4.11.1	Summary of graph pattern matching.	53
4.11.2	Paths.	53
4.11.3	Path patterns.	54
4.11.4	Graph pattern variables.	55
4.11.5	References to graph pattern variables.	56
4.11.6	Path pattern matching.	57
4.11.7	Path modes.	58
4.11.8	Selective path search prefixes.	59
4.11.9	Match modes.	59
4.12	Data types.	59
4.12.1	General introduction to data types and base types.	59
4.12.2	Major classes of data types.	60
4.12.3	Data type descriptors.	62
4.12.4	Naming of data types and base types.	62
4.12.5	Material, nullable, and immaterial data types.	63
4.12.6	Most specific static value type and static base type.	63
4.12.7	Open and closed data types.	63
4.12.8	Additional terminology related to data types.	64
4.13	GQL-object types.	64
4.13.1	Introduction to GQL-object types and related base types.	64
4.13.2	Graph types and graph element types.	65
4.13.2.1	Introduction to graph types and graph element types.	65
4.13.2.2	Graph type descriptors.	65
4.13.2.3	Node types.	66
4.13.2.4	Edge types.	67
4.13.2.5	Property types.	68
4.13.2.6	Key label sets.	69
4.13.2.7	Structural consistency of element types.	69
4.13.3	Binding table types.	70
4.14	Dynamic union types.	71
4.14.1	Introduction to dynamic union types and the dynamic base type.	71
4.14.2	Dynamic union data type descriptors.	71
4.14.3	Characteristics of dynamic union types.	71
4.14.4	Dynamic generation of type tests and casts.	72
4.14.4.1	Introduction to dynamic generation of type tests and casts for <value expression>s.	72
4.14.4.2	Dynamic generation of type tests and strict casts for a <value expression> without operands.	72
4.14.4.3	Dynamic generation of type tests and strict casts for a <value expression> with operands.	73
4.14.4.4	Dynamic generation of additional type tests and lax casts for a <value expression>.	75
4.15	Constructed value types.	75

4.15.1	Introduction to constructed value types and related base types.	75
4.15.2	Path value types.	76
4.15.3	List value types.	76
4.15.4	Record types.	77
4.16	Predefined value types.	79
4.16.1	Introduction to predefined value types and related base types.	79
4.16.2	Boolean types.	81
4.16.3	Character string types.	82
4.16.3.1	Introduction to character strings.	82
4.16.3.2	Collations.	83
4.16.4	Byte string types.	84
4.16.5	Numeric types.	85
4.16.5.1	Introduction to numbers.	85
4.16.5.2	Characteristics of numbers.	85
4.16.5.3	Binary exact numeric types.	87
4.16.5.4	Decimal exact numeric types.	88
4.16.5.5	Approximate numeric types.	89
4.16.6	Temporal types.	90
4.16.6.1	Introduction to temporal data.	90
4.16.6.2	Temporal instant types.	90
4.16.6.3	Temporal duration types.	91
4.16.6.4	Operators involving values of temporal types.	92
4.16.7	Reference value types.	93
4.16.8	Immaterial value types: null type and empty type.	94
4.17	Sites.	94
4.17.1	General description of sites.	94
4.17.2	Static and dynamic sites.	95
4.17.3	Assignment and store assignment.	95
4.17.4	Nullability.	95
4.17.4.1	Introduction to nullability.	95
4.17.4.2	Nullability requirements.	96
4.17.4.3	Nullability inference.	96
5	Notation and conventions.	97
5.1	Notation taken from The Unicode® Standard	97
5.2	Notation.	97
5.3	Conventions.	98
5.3.1	Specification of syntactic elements.	98
5.3.2	Use of terms.	99
5.3.2.1	Syntactic containment.	99
5.3.2.2	Keywords and <keyword>s.	100
5.3.2.3	Terms denoting rule requirements.	100
5.3.2.4	Rule evaluation order.	100
5.3.2.5	Conditional rules.	101
5.3.2.6	Syntactic substitution.	102
5.3.3	Descriptors.	102
5.3.4	Subclauses used as subroutines.	103
5.3.5	Document typography.	103

5.3.6	Index typography.....	103
5.3.7	Feature ID and Feature Name.	104
6	<GQL-program>.....	105
7	Session management.....	107
7.1	<session set command>.....	107
7.2	<session reset command>.....	111
7.3	<session close command>.....	113
7.4	<session parameter specification>.....	114
8	Transaction management.....	115
8.1	<start transaction command>.....	115
8.2	<transaction characteristics>.....	116
8.3	<rollback command>.....	117
8.4	<commit command>.....	118
9	Procedure specification.....	119
9.1	<procedure specification>.....	119
9.2	<procedure body>.....	121
10	Variable definitions.....	125
10.1	<graph variable definition>.....	125
10.2	<binding table variable definition>.....	127
10.3	<value variable definition>.....	129
11	Object expressions.....	131
11.1	<graph expression>.....	131
11.2	<binding table expression>.....	133
11.3	<object expression primary>.....	135
12	Catalog-modifying statements.....	136
12.1	<linear catalog-modifying statement>.....	136
12.2	<create schema statement>.....	137
12.3	<drop schema statement>.....	138
12.4	<create graph statement>.....	139
12.5	<drop graph statement>.....	142
12.6	<create graph type statement>.....	143
12.7	<drop graph type statement>.....	145
12.8	<call catalog-modifying procedure statement>.....	146
13	Data-modifying statements.....	147
13.1	<linear data-modifying statement>.....	147
13.2	<insert statement>.....	149
13.3	<set statement>.....	154
13.4	<remove statement>.....	158
13.5	<delete statement>.....	160
13.6	<call data-modifying procedure statement>.....	162
14	Query statements.....	163
14.1	<composite query statement>.....	163
14.2	<composite query expression>.....	164
14.3	<linear query statement> and <simple query statement>.....	168
14.4	<match statement>.....	170

14.5	<call query statement>.....	173
14.6	<filter statement>.....	174
14.7	<let statement>.....	175
14.8	<for statement>.....	177
14.9	<order by and page statement>.....	180
14.10	<primitive result statement>.....	182
14.11	<return statement>.....	185
14.12	<select statement>.....	190
15	Procedure calling.....	199
15.1	<call procedure statement> and <procedure call>.....	199
15.2	<inline procedure call>.....	201
15.3	<named procedure call>.....	203
16	Common elements.....	205
16.1	<at schema clause>.....	205
16.2	<use graph clause>.....	206
16.3	<graph pattern binding table>.....	208
16.4	<graph pattern>.....	213
16.5	<insert graph pattern>.....	219
16.6	<path pattern prefix>.....	222
16.7	<path pattern expression>.....	226
16.8	<label expression>.....	236
16.9	<path variable reference>.....	238
16.10	<element variable reference>.....	239
16.11	<graph pattern quantifier>.....	240
16.12	<simplified path pattern expression>.....	242
16.13	<where clause>.....	247
16.14	<yield clause>.....	248
16.15	<group by clause>.....	250
16.16	<order by clause>.....	252
16.17	<sort specification list>.....	253
16.18	<limit clause>.....	256
16.19	<offset clause>.....	257
17	Object references.....	258
17.1	<schema reference> and <catalog schema parent and name>.....	258
17.2	<graph reference> and <catalog graph parent and name>.....	261
17.3	<graph type reference> and <catalog graph type parent and name>.....	263
17.4	<binding table reference> and <catalog binding table parent and name>.....	264
17.5	<procedure reference> and <catalog procedure parent and name>.....	266
17.6	<catalog object parent reference>.....	267
17.7	<reference parameter specification>.....	269
17.8	<external object reference>.....	271
18	Type elements.....	272
18.1	<nested graph type specification>.....	272
18.2	<node type specification>.....	277
18.3	<edge type specification>.....	281
18.4	<label set phrase> and <label set specification>.....	289

18.5	<property types specification>.....	290
18.6	<property type>.....	291
18.7	<property value type>.....	292
18.8	<binding table type>.....	293
18.9	<value type>.....	294
18.10	<field type>.....	318
19	Predicates.....	319
19.1	<search condition>.....	319
19.2	<predicate>.....	320
19.3	<comparison predicate>.....	321
19.4	<exists predicate>.....	326
19.5	<null predicate>.....	327
19.6	<value type predicate>.....	328
19.7	<normalized predicate>.....	329
19.8	<directed predicate>.....	330
19.9	<labeled predicate>.....	331
19.10	<source/destination predicate>.....	332
19.11	<all_different predicate>.....	334
19.12	<same predicate>.....	335
19.13	<property_exists predicate>.....	336
20	Value expressions and specifications.....	337
20.1	<value expression>.....	337
20.2	<value expression primary>.....	339
20.3	<value specification>.....	340
20.4	<dynamic parameter specification>.....	342
20.5	<let value expression>.....	343
20.6	<value query expression>.....	344
20.7	<case expression>.....	346
20.8	<cast specification>.....	349
20.9	<aggregate function>.....	363
20.10	<element_id function>.....	369
20.11	<property reference>.....	370
20.12	<binding variable reference>.....	372
20.13	<path value expression>.....	375
20.14	<path value constructor>.....	377
20.15	<list value expression>.....	378
20.16	<list value function>.....	379
20.17	<list value constructor>.....	381
20.18	<record constructor>.....	383
20.19	<field>.....	385
20.20	<boolean value expression>.....	386
20.21	<numeric value expression>.....	388
20.22	<numeric value function>.....	390
20.23	<string value expression>.....	397
20.24	<character string function>.....	400
20.25	<byte string function>.....	405
20.26	<datetime value expression>.....	407

20.27	<datetime value function>.....	408
20.28	<duration value expression>.....	414
20.29	<duration value function>.....	418
21	Lexical elements.....	422
21.1	Names and variables.....	422
21.2	<literal>.....	425
21.3	<token>, <separator>, and <identifier>.....	438
21.4	<GQL terminal character>.....	448
22	Additional common rules.....	452
22.1	Annotation of a <GQL-program>.....	452
22.2	Machinery for graph pattern matching.....	455
22.3	Evaluation of a <path pattern expression>.....	460
22.4	Evaluation of a selective <path pattern>.....	465
22.5	Satisfaction of a <label expression> by a label set.....	469
22.6	Application of bindings to evaluate an expression.....	471
22.7	Evaluation of an expression on a group variable.....	475
22.8	Application of bindings to generate a record.....	476
22.9	Resolution of a <simple directory path> from a start directory.....	478
22.10	Store assignment.....	479
22.11	Determination of identical values.....	485
22.12	Determination of distinct values.....	487
22.13	Equality operations.....	489
22.14	Ordering operations.....	490
22.15	Grouping operations.....	491
22.16	Determination of collation.....	492
22.17	Graph-type specific combination of property value types.....	493
22.18	General combination of value types.....	494
22.19	Static combination of value types.....	497
22.20	Determination of value type precedence.....	500
23	GQLSTATUS and diagnostic records.....	505
23.1	GQLSTATUS.....	505
23.2	Diagnostic records.....	510
24	Conformance.....	513
24.1	Introduction to conformance.....	513
24.2	Minimum conformance.....	513
24.3	Conformance to features.....	513
24.4	Requirements for GQL-programs.....	515
24.4.1	Introduction to requirements for GQL-programs.....	515
24.4.2	Claims of conformance for GQL-programs.....	515
24.5	Requirements for GQL-implementations.....	516
24.5.1	Introduction to requirements for GQL-implementations.....	516
24.5.2	Claims of conformance for GQL-implementations.....	516
24.5.3	Extensions and options.....	516
24.6	GQL Flagger.....	516
24.7	Implied feature relationships.....	517
Annex A	(informative) GQL conformance summary.....	522

Annex B (informative) **Implementation-defined elements**.....555

Annex C (informative) **Implementation-dependent elements**.....574

Annex D (informative) **GQL optional feature taxonomy**.....577

Annex E (informative) **Maintenance and interpretation of GQL**.....587

Bibliography.....588

Index.....589

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

Tables

Table	Page
1 Valid operators involving values of temporal types.	92
2 Symbols used in BNF.	97
3 Conversion of simplified syntax delimiters to default edge delimiters.	244
4 Valid combinations of source and target and types.	350
5 Truth table for the AND Boolean operator.	387
6 Truth table for the OR Boolean operator.	387
7 Truth table for the IS Boolean operator.	387
8 GQLSTATUS class and subclass codes.	505
9 Operation codes.	510
10 Implied feature relationships.	517
D.1 Feature taxonomy for optional features.	577

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

Figures

Figure		Page
1	Components of a GQL-environment.	21
2	Components of a GQL-catalog.	25
3	Major classes of data types.	61

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

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 technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see <https://www.iso.org/directives> or https://www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at <https://www.iso.org/patents> and <https://patents.iec.ch>. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see <https://www.iso.org/iso/foreword.html>. In the IEC, see <https://www.iec.ch/under-standing-standards>.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

Any feedback or questions on this document should be directed to the user's national standards body. <https://www.iso.org/members.html> and <https://www.iec.ch/national-committees>.

Introduction

This document defines GQL, a database language for modeling structured data as a graph, and for storing, querying, and modifying that data in a graph database or other graph store. There are two major graph data models in current use: the Resource Description Framework (RDF) model and the Property Graph model. The RDF model has been standardized by W3C in a number of specifications. GQL addresses the Property Graph model.

Property graphs organize data as entities called nodes (or, alternatively, vertices) and edges (or, alternatively, relationships). Each graph element (a node or an edge) can have associated labels and properties. The flexibility and intuitiveness of the data model and its emphasis on interconnections between graph elements make property graphs suitable for storing complex knowledge and for analytical tasks such as entity resolution, fraud detection, cyber security, and forecasting.

GQL is declarative and transactional, taking inspiration from SQL and from leading independently-developed property graph languages. Property graphs select data primarily through path pattern matching. Defining path pattern searches in a graph is often simpler or more flexible than defining the equivalent joins in SQL. The flexible data model, the availability of path pattern matching, and the efficiency of traversing edges compared to joining tables have led to increasing interest in property graph databases.

Various graph data models have been around for many decades, but it is only since the early 21st century that the demand has driven the rise of commercial graph database and graph analytical systems for property graphs.

GQL provides a standard yet flexible common language for this growing market. GQL supports the same graph pattern matching syntax as SQL Property Graph Queries, ISO/IEC 9075-16, Information technology — Database languages SQL — Part 16: Property Graph Queries (SQL/PGQ). While SQL/PGQ provides the property graph data model and graph pattern matching on top of a relational SQL database, GQL is intended for pure property graphs that provide graph data management independent from SQL.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

Information technology — Database languages — GQL

1 Scope

This document defines data structures and basic operations on property graphs. It provides capabilities for creating, accessing, querying, maintaining, and controlling property graphs and the data they comprise.

This document specifies the syntax and semantics of a data management language for specifying and modifying the structure of property graphs and collections thereof. This document provides a vehicle for portability of data definitions and manipulation among GQL-implementations.

Implementations of this document can exist in environments that also support application programming languages, end-user query facilities, and various tools for database design, data administration, and performance optimization.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 8601-1:2019, *Date and time — Representations for information interchange — Part 1: Basic rules*

ISO 8601-2:2019, *Date and time — Representations for information interchange — Part 2: Extensions*

ISO/IEC 9075-2:2023, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

ISO/IEC 14651:2020, *Information technology — International string ordering and comparison — Method for comparing character strings and description of the common template tailorable ordering*

IEEE Std 754:2019, *IEEE Standard for Floating-Point Arithmetic*

Internet Engineering Task Force (IETF) RFC 3986, *Uniform Resource Identifier (URI): Generic Syntax*. Edited by: Berners-Lee, T., Fielding, R., Masinter, L. January 2005. Available at: <https://www.ietf.org/rfc/rfc3986.txt>

Kuhn, Markus. *Coordinated Universal Time with Smoothed Leap Seconds (UTC-SLS)* [online]. University of Cambridge: IETF, January 2006. Available at <https://tools.ietf.org/html/-draft-kuhn-leapsecond-00>

The Unicode Consortium. *The Unicode Standard* (Information about the latest version of the Unicode standard can be found by using the "Latest Version" link on the "Enumerated Versions of The Unicode Standard" page.) [online]. Mountain View, California, USA: The Unicode Consortium, Available at <https://www.unicode.org/versions/enumeratedversions.html>

The Unicode Consortium. *Unicode Collation Algorithm* [online]. Mountain View, California, USA: The Unicode Consortium, Available at <https://www.unicode.org/reports/tr10/>

The Unicode Consortium. *Unicode Normalization Forms* [online]. Mountain View, California, USA: The Unicode Consortium, Available at <https://www.unicode.org/reports/tr15/>

The Unicode Consortium. *Unicode Identifier and Pattern Syntax* [online]. Mountain View, California, USA: The Unicode Consortium, Available at <https://www.unicode.org/reports/tr31/>

van Kesteren, A. *URL Living Standard* [online]. [Place of publication unknown]: WHATWG, Available at <https://url.spec.whatwg.org>

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <https://www.electropedia.org/>

3.1 General terms and definitions

3.1.1

atomic, adj.

incapable of being subdivided

Note 1 to entry: The antonym is *composite* (3.1.2).

3.1.2

composite, adj.

comprising distinguishable elements

Note 1 to entry: The antonym is *atomic* (3.1.1).

3.1.3

identify, verb

reference without ambiguity

3.1.4

implementation-defined, adj.

possibly differing between GQL-implementations, but specified by the implementer for each particular GQL-implementation

3.1.5

implementation-dependent, adj.

possibly differing between GQL-implementations, but not required to be specified by the implementer for any particular GQL-implementation

3.1.6

identifier

value (3.8.1) by which something is identified

3.1.7

system-generated identifier

unique *identifier* (3.1.6) that is different from any user-provided identifier

Note 1 to entry: In this document, system-generated identifiers are not exposed to the user. See [Syntax Rule 23](#)) of [Subclause 21.3](#), “<token>, <separator>, and <identifier>” for detailed provisions regarding the construction of system-generated identifiers.

3.1.8

object

thing that is separately identifiable

Note 1 to entry: Objects are not *values* (3.8.1).

3.1.9

descriptor

coded description of the metadata of a *GQL-object* (3.2.15)

Note 1 to entry: See Subclause 5.3.3, “Descriptors”.

3.1.10

persistent, adj.

continuing to exist indefinitely, until destroyed deliberately

3.1.11

dictionary

<“X” dictionary> mapping defined by a set of pairs comprising an identifier and an “X”

Note 1 to entry: Different pairs never have equivalent identifiers. If a dictionary *DICT* contains the pair (*ID*, *X*), then every identifier that is equivalent to *ID* identifies *X* in *DICT*.

3.1.12

hold, verb

<“X” in a dictionary> contain one identifier-value pair whose value is “X”

3.1.13

set

unordered collection of distinguishable elements

3.1.14

sequence

list

ordered collection of elements that are not necessarily distinguishable

3.1.15

cardinality

<collection> number of elements in that collection

Note 1 to entry: Those elements need not necessarily have distinguishable values. The objects to which this concept applies include graphs, binding tables and the values of constructed types.

3.1.16

temporary, adj.

lasting for only a limited period of time

3.1.17

variable

<“X” variable> *identifier* (3.1.6) assigned to a collection of related *sites* (3.6.8)

Note 1 to entry: A variable represents sites.

3.2 GQL-environment terms and definitions

3.2.1

GQL-environment

milieu in which the *GQL-catalog* (3.3.1) and *GQL-data* (3.3.4) exist and *GQL-requests* (3.2.10) are executed

Note 1 to entry: See Subclause 4.2, “GQL-environments and their components”.

ISO/IEC 39075:2024(en)
3.2 GQL-environment terms and definitions

3.2.2

GQL-server

processor capable of executing a *GQL-request* (3.2.10) that was submitted by a *GQL-client* (3.2.4) and delivering the outcome of that *execution* (3.5.6) back to the GQL-client in accordance with the rules and definitions of the GQL language

Note 1 to entry: See Subclause 4.2.3.3, “GQL-servers”.

3.2.3

principal

object (3.1.8) that represents a user within a GQL-implementation

Note 1 to entry: See Subclause 4.2.4.1, “Principals”.

3.2.4

GQL-client

processor capable of establishing a connection to a *GQL-server* (3.2.2) on behalf of a *GQL-agent* (3.2.5) that is authenticated to represent a *principal* (3.2.3)

Note 1 to entry: See Subclause 4.2.3.2, “GQL-clients”.

3.2.5

GQL-agent

independent process that causes the *execution* (3.5.6) of *procedures* (3.5.21) and *commands* (3.5.23)

Note 1 to entry: See Subclause 4.2.2, “GQL-agents”.

3.2.6

GQL-session

consecutive series of *GQL-requests* (3.2.10) issued on behalf of a single user by the *GQL-agent* (3.2.5) via the *GQL-client* (3.2.4) of the same *GQL-environment* (3.2.1)

Note 1 to entry: See Subclause 4.5, “GQL-sessions”.

3.2.7

session context

context associated with a *GQL-session* (3.2.6) in which multiple *GQL-requests* (3.2.10) are executed consecutively

Note 1 to entry: See Subclause 4.5.2, “Session contexts”.

3.2.8

current session context

session context (3.2.7) associated with the *GQL-session* (3.2.6) of the currently executing *GQL-request* (3.2.10)

3.2.9

session parameter

general parameter (3.2.13) that is defined in a *session context* (3.2.7)

Note 1 to entry: See Subclause 4.10.4.2, “Binding variables and general parameters”.

3.2.10

GQL-request

GQL-program and GQL-request parameters

Note 1 to entry: See Subclause 4.7, “GQL-requests and GQL-programs”.

ISO/IEC 39075:2024(en)
3.2 GQL-environment terms and definitions

3.2.11

GQL-request context

context that augments a *session context* (3.2.7) in which an individual *GQL-request* (3.2.10) is executed

Note 1 to entry: See Subclause 4.7.2, “GQL-request contexts”.

3.2.12

dynamic parameter

general parameter (3.2.13) that is defined in a *GQL-request context* (3.2.11)

Note 1 to entry: See Subclause 4.10.4.2, “Binding variables and general parameters”.

3.2.13

general parameter

pair comprising a name and a pair comprising a *value* (3.8.1) and a *value type* (3.9.17)

Note 1 to entry: The parameter value type of a general parameter is a static value type of the parameter value of that general parameter.

3.2.14

execution stack

push-down stack of *execution contexts* (3.5.8) associated with a *GQL-request* (3.2.10)

3.2.15

GQL-object

object (3.1.8) capable of being manipulated directly by the *execution* (3.5.6) of a *GQL-request* (3.2.10)

Note 1 to entry: Every GQL-object specified in this document has a definition, and possibly, content. See Subclause 4.3, “GQL-objects”.

3.2.16

data object

GQL-object (3.2.15) comprising data

3.2.17

primary object

independently definable *GQL-object* (3.2.15)

Note 1 to entry: A primary object may be defined separately from, but may also be contained as a component of another object.

3.2.18

secondary object

GQL-object (3.2.15) necessarily defined as a component of other *GQL-objects* (3.2.15)

3.3 GQL-catalog terms and definitions

3.3.1

GQL-catalog

persistent (3.1.10), hierarchically-organized collection of *GQL-directories* (3.3.3) and *GQL-schemas* (3.3.5)

Note 1 to entry: See Subclause 4.2.5, “GQL-catalog”.

3.3.2

GQL-catalog root

GQL-directory (3.3.3) or a *GQL-schema* (3.3.5) that is the root of the *GQL-catalog* (3.3.1)

ISO/IEC 39075:2024(en)
3.3 GQL-catalog terms and definitions

3.3.3

GQL-directory

persistent (3.1.10) *dictionary* (3.1.11) of *GQL-directories* (3.3.3) and *GQL-schemas* (3.3.5)

Note 1 to entry: See Subclause 4.2.5.2, “GQL-directories”.

3.3.4

GQL-data

data that is under the control of a GQL-implementation in a *GQL-environment* (3.2.1)

Note 1 to entry: See Subclause 4.2.6, “GQL-data”.

3.3.5

GQL-schema

persistent (3.1.10) *dictionary* (3.1.11) of *primary catalog objects* (3.3.7)

Note 1 to entry: See Subclause 4.2.5.3, “GQL-schemas”.

3.3.6

catalog object

GQL-object (3.2.15) defined, possibly transitively, in the *GQL-catalog* (3.3.1)

3.3.7

primary catalog object

GQL-object (3.2.15) that is both a *primary object* (3.2.17) and a *catalog object* (3.3.6)

3.3.8

visible

<GQL-object> capable of being referenced according to effective access control rules

3.3.9

home schema

default *GQL-schema* (3.3.5) associated with a *principal* (3.2.3)

3.3.10

home graph

default *graph* (3.4.1) associated with a *principal* (3.2.3)

3.4 Graph terms and definitions

3.4.1

graph

property graph

data *object* (3.1.8) comprising zero or more *nodes* (3.4.10) and zero or more *edges* (3.4.11)

Note 1 to entry: In the context of working with property graphs, the term graph is commonly used in the real world as a shorthand for property graph. Without judgment or prejudice, this document prefers the term graph. Additionally, the term property graph is only used to establish the right context in introductory text.

Note 2 to entry: See Subclause 4.3.5, “Graphs”.

3.4.2

multigraph

graph (3.4.1) that allows more than one *edge* (3.4.11) connecting two *nodes* (3.4.10)

3.4.3

directed graph

graph (3.4.1) in which every *edge* (3.4.11) is directed

Note 1 to entry: The antonym is *undirected graph* (3.4.4).

3.4.4

undirected graph

graph (3.4.1) in which every *edge* (3.4.11) is an *undirected edge* (3.4.13)

Note 1 to entry: The antonym is *directed graph* (3.4.3).

3.4.5

mixed graph

graph (3.4.1) that allows both *directed edges* (3.4.12) and *undirected edges* (3.4.13)

3.4.6

empty graph

graph (3.4.1) with zero *nodes* (3.4.10) and zero *edges* (3.4.11)

3.4.7

path

sequence (3.1.14) of an odd number of *graph elements* (3.4.9)

Note 1 to entry: A path always starts and ends with a *node* (3.4.10) and alternates between nodes and *edges* (3.4.11) such that each edge resides in the path between its *endpoints* (3.4.14). See Subclause 4.11.2, "Paths", for a fuller discussion of paths.

Note 2 to entry: A path may comprise a single node.

Note 3 to entry: A node or an edge may be contained multiple times in a path, including via self-loops.

3.4.8

subpath

path (3.4.7) fully contained in another path

Note 1 to entry: A subpath may be identical to its containing path.

3.4.9

graph element

node (3.4.10) or *edge* (3.4.11)

3.4.10

node

vertex

fundamental unit of which a *graph* (3.4.1) is formed

Note 1 to entry: Plural: nodes or vertices.

Note 2 to entry: Both terms, node and vertex, are used in the real world to denote the same concept. Without judgment or prejudice, this document uses only the term node. In BNF productions, wherever the keyword NODE is allowed, the keyword VERTEX can be used instead.

3.4.11

edge

relationship

connection between two *nodes* (3.4.10)

Note 1 to entry: Both terms, edge and relationship, are used in the real world to denote the same concept. Without judgment or prejudice, this document uses only the term edge. In BNF productions, wherever the keyword EDGE is allowed, the keyword RELATIONSHIP can be used instead.

ISO/IEC 39075:2024(en)
3.4 Graph terms and definitions

3.4.12

directed edge

edge (3.4.11) that distinguishes one of its *endpoints* (3.4.14) as its *source node* (3.4.15) and one of its endpoints as its *destination node* (3.4.16)

Note 1 to entry: A directed edge expresses a relationship that is asymmetric.

Note 2 to entry: The antonym is *undirected edge* (3.4.13).

3.4.13

undirected edge

edge (3.4.11) that does not distinguish between its *endpoints* (3.4.14)

Note 1 to entry: An undirected edge expresses a relationship that is necessarily symmetric.

Note 2 to entry: The antonym is *directed edge* (3.4.12).

3.4.14

endpoint

incident node

<edge> one of the two *nodes* (3.4.10) connected by an *edge* (3.4.11)

Note 1 to entry: Both endpoints of an edge may be the same node.

3.4.15

source node

start node

node (3.4.10) that is distinguished as the source of a *directed edge* (3.4.12)

3.4.16

destination node

end node

node (3.4.10) that is distinguished as the destination of a *directed edge* (3.4.12)

3.4.17

label

identifier (3.1.6) associated with a *graph* (3.4.1), a *node* (3.4.10), or an *edge* (3.4.11)

3.4.18

property

<GQL-object> pair comprising a name and a *value* (3.8.1)

Note 1 to entry: See Subclause 4.3.4, "Properties and supported property value types".

3.5 Procedure and command terms and definitions

3.5.1

side effect

change caused during the *execution* (3.5.6) of a *GQL-request* (3.2.10) that is detectable by the execution of another *operation* (3.5.7) as part of the execution of the same or another GQL-request

3.5.2

catalog-modifying, adj.

<side effect> modifying the *GQL-catalog* (3.3.1)

3.5.3

session-modifying, adj.

<side effect> modifying the *GQL-session* (3.2.6) and its context

ISO/IEC 39075:2024(en)
3.5 Procedure and command terms and definitions

3.5.4

transaction-modifying, adj.

<side effect> manipulating GQL-transactions

3.5.5

data-modifying, adj.

<side effect> modifying the content of *data objects* (3.2.16)

3.5.6

execution

computation of a result that may cause *side effects* (3.5.1)

3.5.7

operation

identifiable action carried out by an *execution* (3.5.6)

Note 1 to entry: See Subclause 4.7.7, "Operations".

3.5.8

execution context

context comprising objects that are associated with and manipulated by the *execution* (3.5.6) of *operations* (3.5.7)

Note 1 to entry: See Subclause 4.8, "Execution contexts".

3.5.9

current execution context

execution context (3.5.8) of the currently executing *operation* (3.5.7) of the currently executing *procedure* (3.5.21), *command* (3.5.23), *statement* (3.5.28), or evaluation of an expression

3.5.10

execution outcome

component of an *execution context* (3.5.8) representing the outcome of an *execution* (3.5.6)

Note 1 to entry: See Subclause 4.8.4, "Execution outcomes".

3.5.11

current execution outcome

execution outcome (3.5.10) of the *current execution context* (3.5.9)

3.5.12

successful outcome

<execution context> *execution outcome* (3.5.10) recording a result

Note 1 to entry: A successful outcome records no exception conditions.

Note 2 to entry: The antonym is *failed outcome* (3.5.13).

Note 3 to entry: See Subclause 4.8.4, "Execution outcomes".

3.5.13

failed outcome

<execution context> *execution outcome* (3.5.10) recording an exception condition

Note 1 to entry: A failed outcome records no result.

Note 2 to entry: The antonym is *successful outcome* (3.5.12).

Note 3 to entry: See Subclause 4.8.4, "Execution outcomes".

3.5.14

regular result

result that is a *value* (3.8.1) or a *binding table* (3.8.13) produced by the successful *execution* (3.5.6) of an *operation* (3.5.7)

Note 1 to entry: See Subclause 4.8.4, "Execution outcomes".

3.5.15

omitted result

result indicating the successful *execution* (3.5.6) of an *operation* (3.5.7) that produced no *value* (3.8.1)

Note 1 to entry: The *declared type* (3.9.1) of an omitted result is the empty type.

Note 2 to entry: See Subclause 4.8.4, "Execution outcomes".

3.5.16

result type

data type (3.9.2) of a result

3.5.17

current execution result

result of the *current execution outcome* (3.5.11)

Note 1 to entry: See Subclause 4.8.4, "Execution outcomes".

3.5.18

evaluation

computation of a result that is not permitted to cause *side effects* (3.5.1)

3.5.19

catalog-modifying procedure

procedure (3.5.21) whose *execution* (3.5.6) can cause *catalog-modifying* (3.5.2) *side effects* (3.5.1) only or catalog-modifying side effects together with *data-modifying* (3.5.5) side effects

3.5.20

data-modifying procedure

procedure (3.5.21) whose *execution* (3.5.6) can cause *data-modifying* (3.5.5) *side effects* (3.5.1) only

3.5.21

procedure

description of a computation on input arguments whose *execution* (3.5.6) computes an *execution outcome* (3.5.10) and optionally causes *side effects* (3.5.1)

Note 1 to entry: See Subclause 4.10.2, "Procedures".

3.5.22

query procedure

procedure (3.5.21) whose *execution* (3.5.6) does not cause any *side effects* (3.5.1)

3.5.23

command

operation (3.5.7) that is not specified by a *GQL-procedure* (3.5.26)

Note 1 to entry: See Subclause 4.10.3, "Commands".

3.5.24

session command

command (3.5.23) that can only perform *session-modifying* (3.5.3) *side effects* (3.5.1)

ISO/IEC 39075:2024(en)
3.5 Procedure and command terms and definitions

3.5.25

transaction command

command (3.5.23) that can only perform *transaction-modifying* (3.5.4) *side effects* (3.5.1)

3.5.26

GQL-procedure

procedure (3.5.21) written in the GQL language

3.5.27

external procedure

procedure (3.5.21) provided via an implementation-defined mechanism

3.5.28

statement

operation (3.5.7) executed as part of executing a *procedure* (3.5.21) that updates the *current execution context* (3.5.9) and its *current execution outcome* (3.5.11) and that can cause *side effects* (3.5.1)

3.6 General syntax terms and definitions

3.6.1

scope

<name> one or more BNF non-terminal symbols within which a name is effective

3.6.2

scope

<working object> one or more BNF non-terminal symbols within which a working object is available

3.6.3

scope clause

<name> BNF non-terminal symbol used in defining the *scope* (3.6.1) of the introduced name

3.6.4

scope clause

<working object> BNF non-terminal symbol used in defining the *scope* (3.6.2) of the declared working object

3.6.5

namespace

classification scheme that permits a given name to identify multiple objects that are distinguished by context

3.6.6 assignment

operation whose effect is to ensure that the value at a *site* (3.6.8) *T* (known as the target) is identical to a given value *S* (known as the source) or a representative of *S* obtained using relevant type conversions

Note 1 to entry: There are two kinds of assignment, regular assignment and store assignment. Assignment is regular assignment by default and frequently indicated by using the phrases “*T* is assigned to *S*”, “the value of *T* is assigned to *S*”, or “the value of *S* assigned from *T*”. Store assignment is indicated by explicitly calling the General Rules of Subclause 22.10, “Store assignment”. See Subclause 4.17.3, “Assignment and store assignment”.

Note 2 to entry: If *S* is not included in the *declared type* (3.9.1) *DT* of *T*, then the result of converting *S* to *DT* obtained using relevant type conversions is assigned to *T* (instead of *S*). See Subclause 4.17.3, “Assignment and store assignment”.

Note 3 to entry: GQL-objects are inserted or removed (e.g., into the GQL-schema or other GQL-objects) but never assigned. However, reference values to GQL-objects are subject to provisions regarding assignment.

3.6.7 instance

<“X” instance> physical representation of an “X”

Note 1 to entry: Each instance is at exactly one *site* (3.6.8). An “X” instance has a type that is the type of “X”.

3.6.8 site

place occupied by an *instance* (3.6.7) of a specified data type

Note 1 to entry: See Subclause 4.17, “Sites”.

3.6.9 pattern

syntax for specifying requirements on the on the bindings of variables

3.6.10 pattern variable

<“X” pattern variable> “X” *variable* (3.1.17) that is declared in an “X” *pattern* (3.6.9)

3.6.11 pattern match

pattern variable (3.6.10) bindings fulfilling the requirements imposed by a *pattern* (3.6.9)

3.6.12 linear composition

composition of a *sequence* (3.1.14) of suboperations that forms a *composite* (3.1.2) *operation* (3.5.7) that effectively executes the suboperations in the order given by the sequence

Note 1 to entry: The linear composition of operations over binding tables corresponds to a left lateral join between those tables.

3.6.13 whitespace

sequence (3.1.14) of one or more characters that have the Unicode property White_Space

Note 1 to entry: Whitespace is typically used to separate <non-delimiter token>s from one another, and is always permitted between two tokens in text written in the GQL language.

Note 2 to entry: See Subclause 21.3, “<token>, <separator>, and <identifier>”.

3.7 Graph pattern terms and definitions

3.7.1

edge variable

element variable (3.7.2) that is declared in an *edge pattern* (3.7.11)

Note 1 to entry: An edge variable may be bound to a list of *edges* (3.4.11).

3.7.2

element variable

primary variable

pattern variable (3.6.10) that may be bound to a list of *graph elements* (3.4.9)

Note 1 to entry: An element variable is either a *node variable* (3.7.4) or an *edge variable* (3.7.1).

3.7.3

graph pattern variable

path variable (3.7.5), *subpath variable* (3.7.6) or *element variable* (3.7.2)

3.7.4

node variable

element variable (3.7.2) that is declared in a *node pattern* (3.7.10)

Note 1 to entry: A node variable may be bound to a list of *nodes* (3.4.10).

3.7.5

path variable

pattern variable (3.6.10) that is declared at the head of a <path pattern>

Note 1 to entry: A path variable can be bound to a path binding that is matched by a *path pattern* (3.7.8). See Subclause 22.2, “Machinery for graph pattern matching”, for more details.

Note 2 to entry: The extracted path of the path binding is a *path* (3.4.7).

Note 3 to entry: Extracted path and path binding are defined in Subclause 22.2, “Machinery for graph pattern matching”.

3.7.6

subpath variable

pattern variable (3.6.10) that is declared at the head of a <parenthesized path pattern expression>

3.7.7

graph pattern

set (3.1.13) of one or more *path patterns* (3.7.8)

3.7.8

path pattern

pattern (3.6.9) that matches a *path* (3.4.7)

3.7.9

element pattern

node pattern (3.7.10) or *edge pattern* (3.7.11)

3.7.10

node pattern

path pattern (3.7.8) that matches a single *node* (3.4.10)

3.7.11

edge pattern

path pattern (3.7.8) that matches a single *edge* (3.4.11)

3.7.12

label expression

expression composed from *label* (3.4.17) names using disjunction, conjunction, and negation

Note 1 to entry: Disjunction, conjunction, and negation are denoted respectively by a vertical bar "|", ampersand "&" and exclamation mark "!", with parentheses for grouping. See Subclause 16.8, "<label expression>" for more details.

3.8 Value terms and definitions

3.8.1

value

<"X" value> definite, immutable, and irreducible unit of data of some type "X"

Note 1 to entry: Values stand for themselves. The identity of a value is independent of where it occurs.

Note 2 to entry: Values are not *objects* (3.1.8).

Note 3 to entry: See Subclause 4.4, "Values".

3.8.2

comparable, adj.

<pair of values> capable of being compared

Note 1 to entry: *comparable* (3.9.7) defines the same term for value types.

Note 2 to entry: Whether a pair of values is capable of being compared is either determined by the provisions of Clause 4, "Concepts" regarding value types or by the presence of support for Feature GA04, "Universal comparison". See Subclause 4.4.2, "Comparable values".

3.8.3

identical, adj.

<pair of values> indistinguishable, in the sense that it is impossible, by any means specified in this document, to detect any difference between them

Note 1 to entry: For the full definition, see Subclause 22.11, "Determination of identical values".

3.8.4

distinct, adj.

<pair of values> capable of being distinguished within a given context

Note 1 to entry: Informally, two values are distinct if neither is null and the values are not equal. A null value and a material value are distinct. Two null values are not distinct. See Subclause 4.4.3, "Properties of distinct values", and the General Rules of Subclause 22.12, "Determination of distinct values".

3.8.5

duplicates

two or more values that are not *distinct* (3.8.4)

3.8.6

reference value

value (3.8.1) that describes a globally resolved reference to a *GQL-object* (3.2.15)

Note 1 to entry: See Subclause 4.4.4, "Reference values".

3.8.7

material, adj.

<value> not the *null value* (3.8.8)

3.8.8

null value

special *value* (3.8.1) that is used to indicate the absence of other data

Note 1 to entry: See Subclause 4.4.5, “Material values and the null value”.

3.8.9

binding variable

<execution context> *variable* (3.1.17) assigned to the identified *field* (3.8.15) of the working record or the identified fields of the records of the working table of a given execution context

Note 1 to entry: A binding variable never identifies both a field of the working record and fields of the records of the working table of a given execution context since the record types of the working record and the working table of an execution context are always field name-disjoint.

Note 2 to entry: <binding variable reference>s are resolved to fields of the current working record only.

Note 3 to entry: See Subclause 4.10.4.2, “Binding variables and general parameters”.

3.8.10

graph variable

binding variable (3.8.9) whose *declared type* (3.9.1) is a *graph* (3.4.1) *reference value type* (3.9.18)

3.8.11

binding table variable

binding variable (3.8.9) whose *declared type* (3.9.1) is a *binding table* (3.8.13) *reference value type* (3.9.18)

3.8.12

value variable

binding variable (3.8.9) whose *declared type* (3.9.1) is a *value type* (3.9.17)

3.8.13

binding table

primary object (3.2.17) comprising a collection of zero or more (possibly duplicate) records of the same *record type* (3.9.23)

Note 1 to entry: See Subclause 4.3.6, “Binding tables”, and Subclause 4.13.3, “Binding table types”.

3.8.14

record

value (3.8.1) of a *record type* (3.9.23); possibly empty *set* (3.1.13) of *fields* (3.8.15)

3.8.15

field

<record> pair comprising a name and a value

3.8.16

byte string

element of the byte string type

3.8.17

character string

element of the character string type

3.8.18 collation

process by which, given two strings, it is determined whether the first one is less than, equal to, or greater than the second one

Note 1 to entry: See Subclause 4.16.3.2, “Collations”.

[SOURCE: ISO/IEC 14651:2020, 3.7]

3.9 Type terms and definitions

3.9.1 declared type

unique *data type* (3.9.2) common to every *instance* (3.6.7) that may be assigned to a given *site* (3.6.8) or that can result from *execution* (3.5.6) or *evaluation* (3.5.18) of the *operation* (3.5.7) specified by a given BNF non-terminal instance

Note 1 to entry: For every BNF non-terminal instance *NT* has an execution outcome, if and only if *NT* has an omitted result, the declared type of *NT* is the empty type. See Subclause 4.16.8, “Immaterial value types: null type and empty type”.

Note 2 to entry: See Subclause 4.17.4, “Nullability”, regarding additional provisions for determining the nullability of declared types.

3.9.2 data type

set (3.1.13) of elements with shared characteristics representing data

Note 1 to entry: A data type characterizes the sites that may be occupied by instances of its elements. See Subclause 4.12, “Data types”.

3.9.3 supertype

<“X” supertype> containing all elements of some data type or base type “X”

Note 1 to entry: If *T1* is a supertype of *T2* and *T1* and *T2* are not compatible, then *T1* is a proper supertype of *T2* (“compatible” is defined in Subclause 4.12.7, “Open and closed data types”).

Note 2 to entry: The antonym is *subtype* (3.9.4).

3.9.4 subtype

<“X” subtype> comprising only elements contained in some data type or base type “X”

Note 1 to entry: If *T2* is a subtype of *T1* and *T1* and *T2* are not compatible, then *T2* is a proper subtype of *T1* (“compatible” is defined in Subclause 4.12.7, “Open and closed data types”).

Note 2 to entry: The antonym is *supertype* (3.9.3).

3.9.5 base type

set of *data types* (3.9.2) with shared characteristics

Note 1 to entry: See Subclause 4.12.1, “General introduction to data types and base types”.

ISO/IEC 39075:2024(en)
3.9 Type terms and definitions

3.9.6

assignable, adj.

<of data type, taken pairwise> characteristic of a value type *T1* that permits a value of *T1* to be assigned to a site of a specified value type *T2* using relevant type conversions, where *T1* and *T2* may be the same value type

Note 1 to entry: By default, provisions on the *assignment* (3.6.6) of values of individual value types and related type conversions are considered, as specified in Subclause 4.14, “Dynamic union types”, Subclause 4.15, “Constructed value types”, and Subclause 4.16, “Predefined value types”.

3.9.7

comparable, adj.

<pair of value types> comprising only pairwise comparable values

Note 1 to entry: *comparable* (3.8.2) defines the same term for values. See Subclause 4.42, “Comparable values”.

3.9.8

constructed, adj.

<data type> comprising *composite* (3.1.2) elements

3.9.9

predefined, adj.

<data type> *atomic* (3.9.10) and provided by the GQL-implementation

3.9.10

atomic, adj.

<data type> comprising only values that are not composed of values of other data types

3.9.11

material, adj.

<data type> excluding the *null value* (3.8.8)

Note 1 to entry: The antonym is *nullable* (3.9.12).

Note 2 to entry: See Subclause 4.12.5, “Material, nullable, and immaterial data types”.

3.9.12

nullable, adj.

<data type> including the *null value* (3.8.8)

Note 1 to entry: The antonym is *material* (3.9.11).

Note 2 to entry: See Subclause 4.12.5, “Material, nullable, and immaterial data types”.

3.9.13

immaterial, adj.

<data type> excluding *material* (3.9.11) values

Note 1 to entry: See Subclause 4.12.5, “Material, nullable, and immaterial data types”.

3.9.14

GQL-object type

data type (3.9.2) comprising *GQL-objects* (3.2.15)

3.9.15

graph type

GQL-object type (3.9.14) describing a *graph* (3.4.1) in terms of restrictions on its labels, properties, nodes, edges, and topology

Note 1 to entry: See Subclause 4.13.2, “Graph types and graph element types”.

3.9.16

binding table type

GQL-object type (3.9.14) of every *binding table* (3.8.13) of a specified *record type* (3.9.23)

Note 1 to entry: See Subclause 4.13.3, “Binding table types”.

3.9.17

value type

data type (3.9.2) comprising *values* (3.8.1)

Note 1 to entry: See Subclause 4.14, “Dynamic union types”, Subclause 4.15, “Constructed value types”, and Subclause 4.16, “Predefined value types”.

3.9.18

reference value type

value type (3.9.17) comprising *reference values* (3.8.6)

3.9.19

supported property value type

non-empty *value type* (3.9.17) that is supported as the type of a *property* (3.4.18) value

Note 1 to entry: See Subclause 4.3.4, “Properties and supported property value types”.

3.9.20

column name

field (3.8.15) name of a *column* (3.9.22)

3.9.21

column type

field (3.8.15) *value type* (3.9.17) of a *column* (3.9.22)

3.9.22

column

field type (3.9.24) of the *record type* (3.9.23) of a *binding table type* (3.9.16)

3.9.23

record type

value type (3.9.17) that describes the *set* (3.1.13) of *fields* (3.8.15) of a *record* (3.8.14) in terms of their *field type* (3.9.24)

Note 1 to entry: See Subclause 4.15.4, “Record types”.

3.9.24

field type

<record type> pair comprising a name and a *value type* (3.9.17)

3.10 Temporal terms and definitions

3.10.1

time

mark attributed to an instant or a time interval on a specified time scale

[SOURCE: ISO 8601-1:2019, 3.1.1.2]

3.10.2

instant

point on the time axis

[SOURCE: ISO 8601-1:2019, 3.1.1.3]

ISO/IEC 39075:2024(en)
3.10 Temporal terms and definitions

3.10.3

time scale

system of ordered marks that can be attributed to instants on the time axis, one instant being chosen as the origin

[SOURCE: ISO 8601-1:2019, 3.1.1.5]

3.10.4

duration

non-negative quantity of time equal to the difference between the final and initial instants of a time interval

[SOURCE: ISO 8601-1:2019, 3.1.1.8]

3.10.5

time of day

time occurring within a calendar day

[SOURCE: ISO 8601-1:2019, 3.1.1.16]

3.10.6

Gregorian calendar

calendar in general use that defines a calendar year that closely approximates the tropical year

[SOURCE: ISO 8601-1:2019, 3.1.1.19]

3.10.7

time shift

constant duration difference between times of two time scales

[SOURCE: ISO 8601-1:2019, 3.1.1.25]

3.10.8

calendar date

particular calendar day represented by its calendar year, its calendar month and its calendar day of month

[SOURCE: ISO 8601-1:2019, 3.1.2.7]

3.10.9

representation with reduced precision

abbreviation of a date and time representation by omission of lower order time scale components

[SOURCE: ISO 8601-1:2019, 3.1.3.7]

3.10.10

negative duration

duration in the reverse direction to the proceeding time scale

[SOURCE: ISO 8601-2:2019, 3.1.1.7]

4 Concepts

4.1 Use of terms

The concepts on which this document is based are described in terms of objects and values.

Some objects are a component of an object on which they depend. If an object *O* ceases to exist, then every object dependent on *O* also ceases to exist. The metadata representation of an object is known as a descriptor. See [Subclause 5.3.3, “Descriptors”](#).

4.2 GQL-environments and their components

4.2.1 General description of GQL-environments

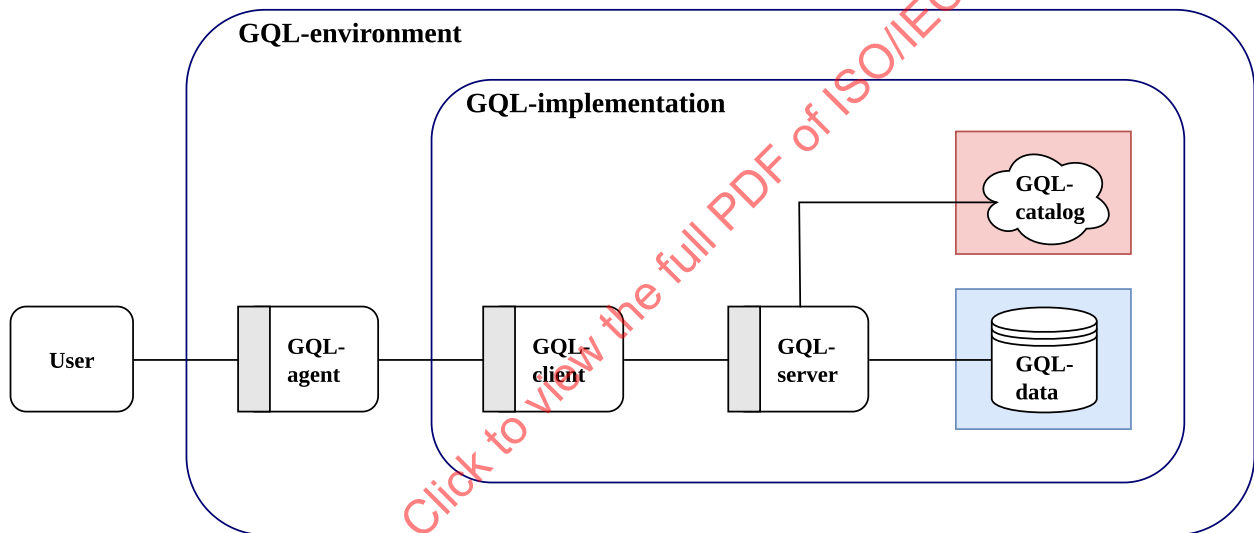


Figure 1 — Components of a GQL-environment

A GQL-environment is a milieu in which the GQL-catalog and GQL-data exist and GQL-requests are executed. A pictorial overview is shown in [Figure 1, “Components of a GQL-environment”](#). A GQL-environment comprises:

- One GQL-agent.
- One GQL-implementation containing one GQL-client and one GQL-server.
- Zero or more authorization identifiers that identify principals.
- One GQL-catalog that comprises one GQL-catalog root. A GQL-catalog that comprises only the GQL-catalog root is considered to be empty. The GQL-catalog is not required to be empty after the instantiation of the GQL-environment.
- The sites, principally catalog objects, that contain GQL-data, as described by the content of the GQL-schemas. This data can be thought of as “the database”, but the term is not used in this document, because it has different meanings in the general context.

ISO/IEC 39075:2024(en)
4.2 GQL-environments and their components

This document recognizes that an installation of a software component that implements GQL can provide multiple GQL-clients, multiple GQL-servers, and multiple GQL-catalogs such that some GQL-agents can use multiple GQL-clients and some GQL-servers can share the same GQL-catalog. In such a scenario, every interaction of one GQL-agent, one GQL-client, one GQL-server, and one GQL-catalog is understood to occur in an isolated GQL-environment. Each such GQL-environment is considered separately in terms of conformance. Any interaction between multiple such GQL-environments is implementation-dependent (UA001) within the constraints of GQL-transaction semantics and is understood as the activity of additional GQL-agents.

4.2.2 GQL-agents

Subclause 3.2, “GQL-environment terms and definitions” defines a GQL-agent as an independent process that causes the execution of procedures and commands. For this purpose, a GQL-agent utilizes an implementation-defined (IW001) mechanism to instruct the GQL-client to submit GQL-requests to the GQL-server within a GQL-session, and thus cause the GQL-implementation to execute the GQL-programs of the submitted GQL-requests.

NOTE 1 — See Subclause 4.7.3, “Execution of GQL-requests”.

4.2.3 GQL-implementations

4.2.3.1 Introduction to GQL-implementations

A *GQL-implementation* is a processor that executes GQL-requests submitted by a GQL-agent. A GQL-implementation, as perceived by a GQL-agent, includes one GQL-client, to which that GQL-agent is bound, and one GQL-server. A GQL-implementation is able to conform to this document even if it allows more than one GQL-server to exist in a GQL-environment.

Because a GQL-implementation is specified only in terms of how it manages GQL-sessions and executes GQL-requests, the concept denotes an installed instance of some software component (database management system). This document does not distinguish between features of the GQL-implementation that are determined by the software implementer and those determined by the installer.

In this document some aspects are marked as “implementation-defined”. These aspects are permitted to differ in different implementations but each conforming implementation is required to fully specify that aspect. See Subclause 24.5.2, “Claims of conformance for GQL-implementations”, for further information.

Each implementation-defined aspect is assigned a code that is placed in parentheses after each occurrence of that implementation-defined aspect in this document. See for example, Syntax Rule 17) of Subclause 18.9, “<value type>”:

“If neither <max length> nor <fixed length> are specified or implicit in a <character string type> *CST*, then the maximum length of the character string type specified by *CST* is implementation-defined (IL013) but shall be greater than or equal to $2^{14} - 1 = 16383$ characters.”

The codes assigned to implementation-defined aspects comprise the letter “I” followed by a letter and three digits.

The codes assigned to implementation-defined aspects are stable and can be depended on to remain constant. The codes themselves have no other meaning than to identify the specific implementation-defined aspect.

For convenience, all the implementation-defined aspects in this document are collected together in the non-normative Annex B, “Implementation-defined elements”, which lists those places in this document where an implementation-defined aspect is referenced.

In this document some aspects are marked as “implementation-dependent”. These aspects are permitted to differ in different implementations, but they are not necessarily specified for any particular GQL-

ISO/IEC 39075:2024(en)
4.2 GQL-environments and their components

implementation. Indeed, a GQL-implementation is not required to exhibit consistent behavior with regard to a given implementation-dependent aspect. Its behavior may depend on aspects such as the chosen access path, which may vary over time. An application should not depend on the specific behavior of any implementation-dependent aspect.

Each implementation-dependent feature is assigned a code that is placed in parentheses after each occurrence of that implementation-dependent feature in this document. See for example, [General Rule 2\)b](#) of [Subclause 20.10](#), “<element_id function>”:

“Otherwise, the result of *EIF* is an implementation-dependent ([UV004](#)) value that encapsulates the identity of the referent of *GRV* for the duration of the currently executing GQL-request.”

The codes assigned to implementation-dependent aspects comprise the letter “U” followed by a letter and three digits.

The codes assigned to implementation-dependent aspects are stable and can be depended on to remain constant. The codes themselves have no other meaning than to identify the specific implementation-dependent aspect.

For convenience, all the implementation-dependent aspects in this document are collected together in the non-normative [Annex C](#), “[Implementation-dependent elements](#)”, which lists those places in this document where an implementation-dependent aspect is referenced.

This document recognizes that there is a possibility that GQL-client and GQL-server software components have been obtained from different implementers; it does not specify the method of interaction or communication between GQL-client and GQL-server.

4.2.3.2 GQL-clients

A GQL-client is a processor capable of establishing a connection to a GQL-server on behalf of a GQL-agent that is authenticated to represent a principal. A GQL-client is perceived by the GQL-agent to be part of the GQL-implementation; a GQL-client establishes and manages the sequence of GQL-sessions between itself and its GQL-server and maintains a GQL-status object and other state data relating to interactions between itself, the GQL-agent, and the GQL-server for its current GQL-session. A GQL-agent submits GQL-requests to a GQL-server via a GQL-client.

A GQL-implementation may detect the loss of the connection between the GQL-client and GQL-server during the execution of a statement. When such a connection failure is detected, an exception condition is raised: *transaction rollback — statement completion unknown (40003)*. This exception condition indicates that the results of the actions performed in the GQL-server on behalf of the GQL-client are unknown to the GQL-agent. Similarly, a GQL-implementation may detect the loss of the connection during the execution of a <commit command>. When such a connection failure is detected, an exception condition is raised: *connection exception — transaction resolution unknown (08007)*. This exception condition indicates that the GQL-implementation cannot verify whether the GQL-transaction was committed successfully, rolled back, or left active.

4.2.3.3 GQL-servers

A GQL-server is a processor capable of executing a GQL-request that was submitted by a GQL-client and delivering the outcome of that execution back to the GQL-client in accordance with the rules and definitions of the GQL language. A GQL-server is perceived by the GQL-agent to be part of the GQL-implementation; a GQL-server manages the GQL-catalog and GQL-data.

The GQL-server of a GQL-environment:

- Manages the sequence of GQL-sessions taking place between itself and the GQL-client on behalf of the GQL-agent.

ISO/IEC 39075:2024(en)
4.2 GQL-environments and their components

- Executes GQL-requests received from the GQL-client to completion and delivers request outcomes back to the GQL-client as required.

NOTE 2 — See Subclause 4.7.3, “Execution of GQL-requests”.

- Maintains the state of the GQL-session, including the authorization identifier, the GQL-transaction, and certain session defaults.

4.2.4 Basic security model

4.2.4.1 Principals

A principal is an implementation-defined (ID001) object that represents a user within a GQL-implementation.

A principal is identified by one or more *authorization identifiers*. The means of creating and destroying authorization identifiers, and their mapping to principals, is implementation-defined (IW002).

NOTE 3 — Allowing multiple authorization identifiers to map to a single principal allows a user to operate with different sets of privileges. See Subclause 4.2.4.2, “Authorization identifiers and privileges”.

Every principal is associated with:

- An optional home schema, which is a GQL-schema.
- An optional home graph, which is a graph.

The manner in which this association is specified is implementation-defined (ID002).

4.2.4.2 Authorization identifiers and privileges

An authorization identifier identifies a principal and a set of privileges for that principal. An authorization identifier is represented by an implementation-defined (IV015) <authorization identifier>.

A *privilege* is a permission to cause certain kinds of operations to be performed by the execution of GQL-requests.

The set of privileges identified by an authorization identifier is implementation-defined (ID003).

If the execution of a GQL-request (See Subclause 4.7, “GQL-requests and GQL-programs”) would perform an operation that is not permitted according to the set of privileges identified by the session authorization identifier, then an exception condition is raised: *syntax error or access rule violation (42000)*.

4.2.5 GQL-catalog

4.2.5.1 General description of the GQL-catalog

The GQL-catalog is a persistent, hierarchically-organized collection of GQL-directories and GQL-schemas.

The GQL-catalog is pictorially represented in Figure 2, “Components of a GQL-catalog”.

The GQL-catalog comprises the GQL-catalog root, which is either a GQL-directory or a GQL-schema. The name of the GQL-catalog root is the zero-length character string.

ISO/IEC 39075:2024(en)
4.2 GQL-environments and their components

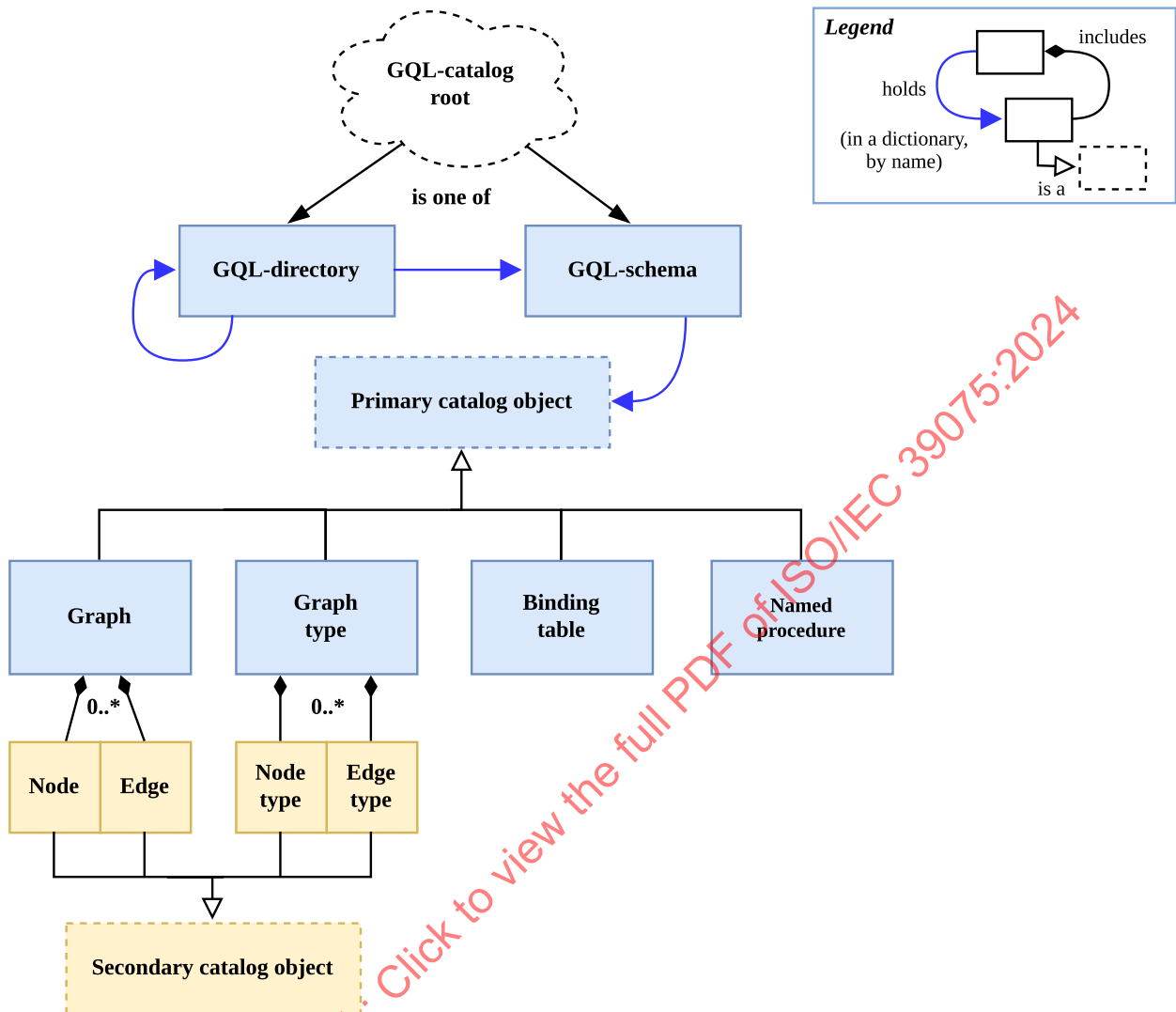


Figure 2 — Components of a GQL-catalog

The maximum depth of nesting of GQL-directories is implementation-defined (IL020). If this depth is zero, then GQL-directories are not supported.

NOTE 4 — This supports GQL-implementations that provide a GQL-catalog that has only a single GQL-schema as its GQL-catalog root, has a single GQL-directory that only contains GQL-schemas as its GQL-catalog root, has a nesting structure that directly matches that of SQL such that its GQL-catalog root is a GQL-directory of SQL-catalogs (represented as GQL-directories) containing GQL-schemas, or has a nesting structure that is suitably restricted in some other way.

4.2.5.2 GQL-directories

A GQL-directory is a persistent dictionary of GQL-directories and GQL-schemas.

The GQL-directory that holds some other GQL-directory or GQL-schema *DOS* is called the *parent directory* of *DOS*. The GQL-catalog root has no parent directory.

Every GQL-directory *DIR* is described by its GQL-directory descriptor that comprises:

ISO/IEC 39075:2024(en)
4.2 GQL-environments and their components

- The set of named subobjects that comprises (identifier, object descriptor) pairs that associate identifiers to the descriptors of every GQL-directory and GQL-schema held by *DIR*.

The name of a GQL-directory *DIR* (also known as *GQL-directory name*) is a character string that is defined as follows. If *DIR* is the GQL-catalog root, then the name of *DIR* is the zero-length character string; otherwise, *DIR* is not the GQL-catalog root and the name of *DIR* is the canonical name form of the identifier that identifies *DIR* within its parent directory.

NOTE 5 — A GQL-directory cannot contain a GQL-directory and a GQL-schema that have equal names.

A GQL-implementation may automatically create a GQL-directory in an implementation-defined (IW015) way.

NOTE 6 — For instance, it is possible for a GQL-implementation to create (recursively) GQL-directories when a GQL-schema is created in a thus far non-existing GQL-directory.

4.2.5.3 GQL-schemas

A GQL-schema is a persistent dictionary of primary catalog objects. Every GQL-schema *SCHEMA* is described by its GQL-schema descriptor that comprises:

- The owner of *SCHEMA*, an authorization identifier of a principal.
- The set of named subobjects of *SCHEMA* that comprises (identifier, object descriptor) pairs that associate identifiers to the catalog object descriptors of every catalog object held by *SCHEMA*.

The name of a GQL-schema (also known as *GQL-schema name*) is a character string that is defined as follows. The name of a GQL-schema that is the GQL-catalog root is the zero-length character string; the name of every GQL-schema *SCHEMA* that is not the GQL-catalog root is the canonical name form of the identifier that identifies *SCHEMA* within its parent directory.

A GQL-implementation may provide some GQL-schemas that cannot be dropped by the execution of a GQL-request.

A catalog object is a GQL-object defined, possibly transitively, in the GQL-catalog. Every catalog object is created directly or indirectly in the context of a GQL-schema, and owned by that GQL-schema.

Every catalog object is described by a catalog object descriptor that is included in the set of named subobjects in the descriptor of the containing GQL-schema or primary catalog object that holds it.

The name of a catalog object (also known as *catalog object name*) is a character string defined as follows. The name of every catalog object *OBJECT* is the canonical name form of the identifier that identifies *OBJECT* within the containing GQL-schema or primary catalog object that holds it.

NOTE 7 — A GQL-schema or primary catalog object cannot hold two different catalog objects with equal names.

A catalog object descriptor is one of:

- A graph descriptor.
- A graph type descriptor.
- A binding table descriptor.
- A named procedure descriptor.

NOTE 8 — A GQL-schema cannot contain a GQL-directory or a GQL-schema, because they are not catalog objects.

A GQL-implementation may automatically populate a GQL-schema upon its creation in an implementation-defined (IW016) way.

4.2.6 GQL-data

GQL-data is data that is under the control of a GQL-implementation in a GQL-environment. GQL-data is described by GQL-schemas in the GQL-catalog.

4.3 GQL-objects

4.3.1 General introduction to GQL-objects

A GQL-object is an object capable of being manipulated directly by the execution of a GQL-request.

Every GQL-object is uniquely identified by one or more effectively immutable *internal object identifiers* within the GQL-environment containing the GQL-object. A GQL-object can be associated with one such internal object identifier that is called its *global object identifier* and that reifies its identity during the execution of a GQL-request. A GQL-object associated with a global object identifier is called *globally identifiable*. The value of an object identifier is implementation-dependent (UV001) and is possibly not accessible to the user. When a GQL-object is copied, the copy is assigned a new global object identifier. Global object identifiers are used for definitional purposes only in order to establish the identity of GQL-objects.

Additionally, a GQL-object can have associated descriptors that define its metadata. This document defines different kinds of GQL-objects. The descriptor of a persistent data object describes a catalog object that has a separate existence as GQL-data. Other descriptors describe GQL-objects that have no existence distinct from their descriptors (at least as far as this document is concerned). Hence there is no loss of precision if, for example, the term “path pattern” is used when “path pattern descriptor” would be more strictly correct.

4.3.2 References to GQL-schemas and GQL-objects

A reference to a GQL-schema or a GQL-object identifies a GQL-schema or a GQL-object, respectively, as the referent of the reference. Resolving a reference determines the referent that it identifies.

A *syntactically resolved reference* is a BNF non-terminal instance that identifies its referent *R* by identifying the static site that *R* occupies. Syntactically resolved references are resolved during the application of Syntax Rules.

A syntactically resolved reference is an instance of one of the following:

- A <schema reference> or an <absolute catalog schema reference>, whose referent is GQL-schema.
- A <graph reference> or a <catalog graph parent and name>, whose referent is a graph.
- A <graph type reference> or a <catalog graph type parent and name>, whose referent is a graph type.
- A <binding table reference> or a <catalog binding table parent and name>, whose referent is a binding table.
- A <procedure reference> or a <catalog procedure parent and name>, whose referent is a procedure.

A *globally resolved reference* identifies its referent *R* by the global object identifier associated with *R* in the GQL-environment. Globally resolved references are resolved during the application of Syntax Rules or General Rules. The globally identifiable GQL-object *O* referenced by a site *S* is represented by a globally resolved reference that identifies *O* in GQL-data. The resolution of such a globally resolved reference that represents *O* at *S* is implicit; referring to *O* at *S* implicitly resolves the representing globally resolved reference.

Subclause 3.8, “Value terms and definitions” defines a reference value as a value that describes a globally resolved reference to a GQL-object.

NOTE 9 — See Subclause 4.4.4, “Reference values”.

NOTE 10 — See Subclause 4.16.7, “Reference value types”.

4.3.3 Primary objects and secondary objects

Every GQL-object is either a *primary* object or a *secondary* object.

A primary object is an independently definable GQL-object. Every primary object is globally identifiable and has a descriptor. Reference values can refer to primary objects. Procedures or commands may define new primary objects in the GQL-catalog, assign reference values referring to primary objects to session parameters in a GQL-session, or bind them to local variables, subject to syntax restrictions. Reference values referring to primary objects may also be passed as GQL-request parameters or in procedure arguments. Furthermore, reference values referring to primary objects may be returned as the result of an execution outcome or generally occur as the result of evaluation.

This document defines the following kinds of primary objects:

- Graphs, as defined in Subclause 4.3.5, “Graphs”.
- Graph types, as defined in Subclause 4.13.2, “Graph types and graph element types”.
- Binding tables, as defined in Subclause 4.3.6, “Binding tables”.
- Named procedures, as defined in Subclause 4.10.2, “Procedures”.

A secondary object is a GQL-object necessarily defined as a component of other GQL-objects. Reference values can refer to secondary objects. A procedure or command may create, modify, delete, or otherwise interact with secondary objects as long as the primary object that contains them has not been deleted. Reference values referring to secondary objects may be set as session parameters, passed as GQL-request parameters or as procedure parameter arguments, bound as local variables, may occur as the result of expression evaluation, or may be returned as the result of an execution outcome.

This document defines the following kinds of secondary objects:

- Nodes, as defined in Subclause 4.3.5.1, “Introduction to graphs”.
- Edges, as defined in Subclause 4.3.5.1, “Introduction to graphs”.
- Node types, as defined in Subclause 4.13.2.3, “Node types”.
- Edge types, as defined in Subclause 4.13.2.4, “Edge types”.

4.3.4 Properties and supported property value types

Certain GQL-objects (e.g., graph elements) can have properties, i.e., a pair comprising a name and a value. A property name is the name and a property value is the value of a property of such a GQL-object.

Every property value is the value of a supported property value type. Subclause 3.9, “Type terms and definitions”, defines a supported property value type as a non-empty value type that is supported as the type of a property value.

NOTE 11 — The empty type is not a supported property value type. See Subclause 4.16.8, “Immaterial value types: null type and empty type”.

NOTE 12 — The set of all value types that are supported property value types is included in the conformance claim of a GQL-implementation or a GQL-program. See Subclause 24.2, “Minimum conformance”.

The *dynamic property value type* is an implementation-defined (IV011) dynamic union type such that each supported property value type is a subtype of the dynamic property value type.

NOTE 13 — See Subclause 4.14, “Dynamic union types” for more details on dynamic union types.

4.3.5 Graphs

4.3.5.1 Introduction to graphs

Graphs are the primary form of data that is queried and manipulated by procedures.

In the GQL language, every graph is a property graph. Therefore, both terms can be used interchangeably, although the shorter term “graph” is preferred in this document.

A graph can be a mixed graph, a multigraph, neither, or both and is a primary object that comprises:

- A set of zero or more globally identifiable nodes. Each node comprises:
 - A node label set that comprises a set of zero or more labels. A label has a name, which is an identifier that is unique within the node.
The minimum cardinality of node label sets is implementation-defined (IL001).
The maximum cardinality of node label sets is implementation-defined (IL001).
 - A node property set that comprises zero or more properties. Each property comprises:
 - Its name, which is an identifier that uniquely identifies the property within the node property set.
NOTE 14 — The names of node labels and of node properties are in separate namespaces. That is, a label and a property can have the same name in a node.
 - Its value, which can be of any supported property value type.
The maximum cardinality of node property sets is implementation-defined (IL002).
- A set of zero or more globally identifiable edges. Each edge comprises:
 - An edge label set that comprises a set of zero or more labels. A label has a name, which is an identifier that is unique within the edge.
The minimum cardinality of edge label sets is implementation-defined (IL001).
The maximum cardinality of edge label sets is implementation-defined (IL001).
 - An edge property set that comprises zero or more properties. Each property comprises:
 - Its name, which is an identifier that uniquely identifies the property within the edge property set.
NOTE 15 — The names of edge labels and of edge properties are in separate namespaces. That is, a label and a property can have the same name in an edge.
 - Its value, which can be of any supported property value type.
The maximum cardinality of edge property sets is implementation-defined (IL002).
 - Two (possibly identical) endpoints, which are nodes contained in the same graph.
 - The indication of whether the edge is a directed edge or an undirected edge (which is also called the *directionality* of the edge).

Additionally, a directed edge identifies one of its endpoints as its source, and the other as its destination. The direction of a directed edge is from its source to its destination. If both endpoints of a directed edge are identical, such an edge is a directed loop on a single graph node.

NOTE 16 — A graph can be a catalog object that has a name, but graphs can exist that have no name, or at least have no name that is required to be visible to a user of a GQL-implementation. <graph initializer>s

(Subclause 10.1, “<graph variable definition>”) are examples of graphs that do not have visible names. Graph variables have names but are not visible in the GQL-catalog.

A graph that has a constraining graph type is said to be closed. A closed graph cannot contain nodes and edges that are of node types and edge types, respectively, that are not specified in the graph’s constraining graph type. A graph that does not have a constraining graph type is said to be open. An open graph does not restrict the types of nodes and edges it can contain.

4.3.5.2 Graph descriptors

A graph is described by a graph descriptor that comprises:

- The optional constraining graph type that is a globally resolved reference to a graph type schema object.

For every graph G that is a catalog object, the graph descriptor of G is a catalog object descriptor. The name of G (also known as graph name) is the catalog object name of G . G is a *named graph*.

In this document, the phrase “the graph type of G ”, where G is a graph, is used to refer to the graph type that is identified by the constraining graph type in the graph descriptor of G .

4.3.6 Binding tables

A binding table is a primary object comprising a collection of zero or more (possibly duplicate) records of the same record type. Binding tables mainly serve as:

- The primary iteration construct that drives the execution of procedures.
- A container that holds intermediary results produced by statements such as the matches found by graph pattern matching.
- An execution result that is returned to the GQL-agent as part of an execution outcome.

The elements of the collection of records of a binding table BT are referred to as the records of BT and the number of such records is referred to as the cardinality of BT . The *unit binding table* is the unordered binding table comprising the unit record. An *empty binding table* is a binding table whose cardinality is 0 (zero).

Every binding table has an associated binding table descriptor that comprises:

- The binding table type of the binding table.

NOTE 17 — Binding table records can contain reference values to primary or secondary objects.

NOTE 18 — See Subclause 4.13.3, “Binding table types”.

- The indication of whether the binding table is *ordered* or *unordered*.

NOTE 19 — If a binding table is ordered, then the order of its records has been determined according to some <sort specification>.

- The preferred column name sequence, which is either “not set” or a permutation of the column names of the binding table type.

NOTE 20 — The preferred column name sequence is metadata tracked in this document to allow implementations to provide a column sequence for a binding table that is returned as a result to the GQL-agent.

Given a binding table BT , the record type of BT is the record type of the binding table type BTT of BT , the column names of BT are the column names of BTT , and the columns of BT are the columns of BTT .

The *effective column name sequence* of a binding table BT is defined as follows. If the preferred column name sequence PCS of BT is defined, then the effective column name sequence of BT is PCS . Otherwise,

the effective column name sequence of *BT* is the sequence of column names of *BT*'s binding table type in ascending order.

In the absence of relevant additional provisions, binding tables are implicitly initialized as unordered and their preferred column name sequence is implicitly initialized as "not set".

A copy of a binding table without the columns identified by a set of names and similar grammatical variants describes a new binding table *NT* whose record type is the record type of a given binding table *T* without the fields identified by a set of names *NAMES* and whose records are the records of *T*, each without the fields identified by *NAMES*. If *T* is ordered, *NT* is ordered; otherwise, *NT* is unordered. If the preferred column name sequence of *T* is defined, then the preferred column name sequence of *NT* is the preferred column name sequence of *T* without names in *NAMES*.

If a General Rule *RULE* refers to the *i*-th record *RECORD* of a table *TABLE*, then

Case:

- 1) If *TABLE* is an ordered binding table, then *RECORD* is the *i*-th record of *TABLE* in the order determined by *TABLE*.
- 2) Otherwise, *TABLE* is an unordered binding table and *RECORD* is the *i*-th record in some sequence of all records of *TABLE* in an implementation-dependent (US001) order determined for each application of *RULE* and any of its subrules.

If the application of a General Rule *RULE* processes all records of a binding table using order-independent terms, then the records are to be processed effectively in an implementation-dependent (US001) order determined for that application of *RULE* and the application of all following rules in the same Subclause. Otherwise, the records are processed effectively in the order determined by the binding table.

The records of a binding table are not modified after the initial construction and population of that binding table. The binding table assigned to a site *S* (such as the current working table) can be replaced by assigning a new binding table to *S* but is never modified after its initial construction.

Let *A* and *B* be two binding tables. If the record types of *A* and *B* are field name-equal or field name-disjoint, then *A* and *B* are *column name-equal* or *column name-disjoint*, respectively. If the record types of *A* and *B* are comparable value types or field type-combinable, then *A* and *B* are *column-comparable* or *column-combinable*, respectively.

NOTE 21 — See Subclause 4.15.4, "Record types", for the definitions of field name-equal, field name-disjoint, field type-comparable, and field type-combinable.

The *combined columns* of two column-combinable binding tables *A* and *B* are given by the combined field types of the record type of *A* and the record type of *B*.

NOTE 22 — See Subclause 4.15.4, "Record types", for the definition of combined field types.

Let *A* and *B* be two records whose most specific static value types are field type-combinable material closed record types. If it holds for each pair of fields F_A from *A* and F_B from *B* with the same name that F_A and F_B have equal values, then *A* and *B* are called *naturally joinable*.

Given two binding tables *A* and *B*, the phrase *B is appended to A* or similar grammatical variants is used to insert one copy of every record of *A* into the collection of records of *B*. Appending only occurs during the initial population of *A*.

NOTE 23 — Appending preserves the identity of *A* as a GQL-object.

The *Cartesian product* between a record *R* and a binding table *T* whose closed record types are field name-disjoint is a new binding table *NT* whose columns are given by the union of the field types of *R* and columns of *T* and that comprises a collection of new records obtained by constructing a new record *NR* for every record *TR* of *T* such that the fields of *NR* are the fields of both *TR* and *R*.

By extension, the Cartesian product between two binding tables $T1$ and $T2$ that are column name-disjoint is a new binding table $T3$ whose columns are given by the union of the columns of $T1$ and $T2$ and that comprises a collection of every record from the Cartesian products between each record of $T1$ with $T2$.

The *natural join* between a record R and a binding table T (and vice versa) whose closed record types are comparable value types is a new binding table NT that comprises the collection of all records obtained by constructing a new record for each record TR from T that is naturally joinable with R by amending R with TR . If T is ordered, NT is ordered; otherwise, NT is unordered. In this document, the columns of such a binding table NT are specified explicitly such that the value type of every column of NT is a supertype of the value type of at least one corresponding field type of R or column of T with an equal name.

NOTE 24 — Amending of records is defined in Subclause 4.15.4, “Record types”.

By extension, the natural join between two binding tables $T1$ and $T2$ that are column-comparable is a new binding table $T3$ that comprises the collection of every record from the natural joins between each record of $T1$ with $T2$. In this document, the columns of such a binding table $T3$ are specified explicitly such that the value type of every column of $T3$ is a supertype of the value type of at least one corresponding column of $T1$ or $T2$ with an equal name.

4.4 Values

4.4.1 General information about values

Values defined in this document are definite objects that either occur at sites to which they have been assigned (such as results, parameter values, procedure arguments, property values, or parts of GQL-objects) or are used to describe characteristics of GQL-objects in their descriptors.

Assigning a value to a site implicitly copies the value.

4.4.2 Comparable values

Subclause 3.8, “Value terms and definitions”, defines that two values are comparable values if they are capable of being compared. For every two values LV and RV it holds that LV and RV are comparable values if and only if one of the following is true:

- 1) The provisions in Clause 4, “Concepts” regarding value types explicitly determine LV and RV as comparable values. If this is the case, then LV and RV are said to be *essentially comparable values*.

NOTE 25 — In this document, every two identical values are essentially comparable values.

- 2) The values LV and RV are not essentially comparable values but the GQL-implementation supports Feature GA04, “Universal comparison”. If this is the case, then LV and RV are said to be *universally comparable values*.

NOTE 26 — In the absence of support for Feature GA04, “Universal comparison”, no two values are universally comparable values.

In summary, LV and RV are comparable values if and only if LV and RV are either essentially comparable values or universally comparable values. Further, the two relations defining comparability of values are disjoint, i.e., no two values are both essentially comparable values and universally comparable values.

4.4.3 Properties of distinct values

Two values are distinct if they are capable of being distinguished within a given context.

Two null values are not distinct.

A null value and a material value are distinct.

If two values *V1* and *V2* are not comparable values, then the determination of whether they are distinct raises an exception condition: *data exception — values not comparable (22G04)*.

NOTE 27 — See Subclause 4.4.2, “Comparable values”.

The application of the General Rules of Subclause 22.12, “Determination of distinct values”, determines whether two values are distinct or not.

4.4.4 Reference values

Subclause 3.8, “Value terms and definitions” defines a reference value as a value that describes a globally resolved reference to a GQL-object.

NOTE 28 — A reference value effectively encapsulates the global object identifier of the GQL-object.

If two reference values have referents of the same static base type, then they are essentially comparable values. Two reference values are equal if and only if they refer to the same referent.

Assigning a reference value to a site implicitly copies the reference value but not its referent.

If a referent of a reference value is deleted or the referent’s descriptor is destroyed while the reference value is still occupying a site, then the reference value is *invalidated*. In this document, deleting the referent of a given reference value is considered an idempotent operation. Accessing any characteristics of a descriptor of the referent or content of the referent of an invalidated reference value raises an exception condition: *data exception — reference value, referent deleted (22G11)*.

Copying an invalidated reference value does not constitute an access to a characteristic of the descriptor of the referent or the value of the referent and does not raise the exception.

4.4.5 Material values and the null value

Every value that is not the null value is a material value. The null value is a special value, sometimes denoted by the keyword NULL. This value differs from material values in the following respects:

- The null value is the only value that is included in multiple data types of different static base types.

NOTE 29 — See Subclause 4.12.1, “General introduction to data types and base types”.

- If the GQL-implementation does not support Feature GV71, “Immaterial value types: null type support”, then the data type of the null value implied by the keyword NULL cannot always be inferred; hence in such a GQL-implementation NULL can be used to denote the null value only in certain contexts, rather than everywhere that a literal is permitted.

NOTE 30 — See Subclause 4.16.8, “Immaterial value types: null type and empty type”.

- Although the null value is neither equal to any other value nor not equal to any other value, it is unknown whether or not it is equal to any given value. In some contexts, multiple null values are treated together; for example, the <group by clause> treats all null values together.

Furthermore, the null value of a Boolean type and the truth value *Unknown* are the same value.

NOTE 31 — See Subclause 4.16.2, “Boolean types”.

4.5 GQL-sessions

4.5.1 General description of GQL-sessions

Subclause 3.2, “GQL-environment terms and definitions” defines a GQL-session as a consecutive series of GQL-requests issued on behalf of a single user by the GQL-agent via the GQL-client of the same GQL-environment. At any one time during a GQL-session, exactly one of these consecutive GQL-requests is being executed and is said to be the *executing GQL-request* of the GQL-session.

A GQL-session is created either explicitly by the GQL-client, on behalf of a GQL-agent or implicitly whenever a GQL-client, on behalf of a GQL-agent, initiates a request to a GQL-server and no GQL-session is current. The GQL-session is terminated, either as the result of an explicit <session close command>, or in the absence of an explicit <session close command>, following the last request from that GQL-client, on behalf of that GQL-agent. The mechanism and rules by which a GQL-implementation determines when the last request has been received are implementation-defined (IW003).

The context of a GQL-session is manipulated by session management commands.

4.5.2 Session contexts

4.5.2.1 Introduction to session contexts

A session context is a context associated with a GQL-session in which multiple GQL-requests are executed consecutively. A session context comprises the following characteristics:

- The authorization identifier, which is a character string.
- The principal identified by the authorization identifier.
- The time zone displacement, which is a character string conforming to the representation specified in ISO 8601-1:2019, 4.3.13, “Time shift”.
- The session schema, which is either “not set” or an <absolute catalog schema reference> that identifies a GQL-schema.
- The session graph, which is either “not set” or a material graph reference value.
- The session parameters, which are represented as a (possibly empty) dictionary of general parameters.
The parameter names of implementation-defined (ID049) default session parameters start with underscore ('_').
- The transaction, which is either “not set” or a GQL-transaction.
- The request context, which is either “not set” or a GQL-request context.
- The termination flag, which is a Boolean value.

NOTE 32 — The termination flag is initially set to *False* but the execution of a <session close command> will set it to *True* to signal that the current session is to be terminated.

The current session context is the session context associated with the GQL-session of the currently executing GQL-request.

In this document the following phrases are used:

- The *session authorization identifier* is the authorization identifier of the current session context.
- The *current principal* is the principal of the current session context.
- The *current home schema* is the home schema of the current principal.
- The *current home graph* is the home graph of the current principal.
- The *current time zone displacement* is the time zone displacement of the current session context.
- The *current session schema* is the session schema of the current session context.
- The *current session graph* is the session graph of the current session context.
- The *current session parameters* are the session parameters of the current session context.

- The *current transaction* is the transaction of the current session context.
- The *current transaction access mode* is the <transaction access mode> specified by the transaction characteristics of the current transaction.
- The *current request context* is the request context of the current session context.
- The *current termination flag* is the termination flag of the current session context.

4.5.2.2 Session context creation

A new session context *SCX* is created and initialized as follows:

NOTE 33 — The provisions specified in Subclause 4.7.3, “Execution of GQL-requests” determine the associated session context of the GQL-session.

- Let *AUTH* be the authorization identifier associated with the GQL-agent. The authorization identifier and the principal of *SCX* are set to *AUTH* and the principal identified by *AUTH*, respectively.
- The time zone displacement of *SCX* is set to the implementation-defined (ID048) default time zone displacement.
- If the current home schema is defined, then the session schema of *SCX* is set to the current home schema; otherwise, the session schema of *SCX* is set to “not set”.
- If the current home graph is defined, then the session graph of *SCX* is set to the current home graph; otherwise, the session graph of *SCX* is set to “not set”.
- The session parameters of *SCX* are set to the implementation-defined (ID049) default session parameters.
- The transaction is set to “not set”.
- The request context is set to “not set”.
- The termination flag of *SCX* is set to *False*.

4.5.2.3 Session context modification

In this document, the characteristics of a session context are only modified after their initialization by:

- Setting the time zone displacement.
- Setting the session schema.
- Setting the session graph.
- Setting the session parameters.
- Setting the transaction.
- Setting the request context upon starting or terminating the execution of a GQL-request.
- Setting the termination flag to *True* to signal that the GQL-session is about to be terminated.

4.6 GQL-transactions

4.6.1 General description of GQL-transactions

A *GQL-transaction* (transaction) is a sequence of executions of procedures that is atomic with respect to recovery. That is to say: either the complete execution of each procedure results in a successful outcome, or the failed execution of any procedures results in having no effect on any objects in the GQL-catalog or on GQL-data.

At any time, there is at most one currently active transaction in a GQL-session. A single transaction may span several consecutive GQL-requests of a GQL-session, but a single GQL-request is always contained within a single transaction. Initiation and termination of a transaction does not affect the GQL-session context except for registering the presence or absence of a current transaction.

For the purposes of this document there is only one currently active transaction, which is the behavior of a system that only supports serializable transactions.

Statements within a procedure effectively execute serially. The state of GQL-data and the GQL-catalog, as affected by a successfully-completed statement, are visible to a successor statement.

Any relaxation of the assumption of the serializable transactional behavior (for example, less restrictive isolation levels) is an implementation-defined (IE004) extension.

A GQL-implementation may define different kinds of GQL-transactions to be initiated for the execution of a catalog-modifying procedure, a data-modifying procedure, or a query procedure.

Catalog-modifying procedures performing statements such as CREATE SCHEMA, CREATE GRAPH, DROP GRAPH, and DROP SCHEMA may require the allocation or release of operating system resources that are not under transaction control in a GQL-implementation. It is implementation-defined (IW025) which and how many catalog-modifying procedures are under transaction control, and which catalog-modifying procedures can be contained in a single transaction.

If a GQL-implementation supports Feature GP18, “Catalog and data statement mixing”, then a GQL-transaction can contain the execution of both catalog-modifying and data-modifying procedures. Without Feature GP18, “Catalog and data statement mixing”, executing catalog-modifying and data-modifying procedures in the same GQL-transaction results in an exception. If permitted, there may be additional implementation-defined (IE006) restrictions, requirements, and conditions. If any such restrictions, requirements, or conditions are violated, then an exception condition or a completion condition warning is raised with an implementation-defined (IE007) class and/or subclass code.

4.6.2 Transaction demarcation

The GQL language includes the following explicit transaction demarcation commands, to be issued by agents in GQL-requests to a GQL-server:

A GQL-transaction can be initiated by a GQL-agent submitting a GQL-request that specifies a <start transaction command>. If no GQL-transaction has been initiated, the commencement of execution of a procedure will initiate a GQL-transaction.

The <start transaction command> allows the specification of the characteristics of the GQL-transaction, such as the mode (read-only or read-write). If a <start transaction command> is submitted when there is already an active GQL-transaction, then an exception condition is raised: *invalid transaction state — active GQL-transaction (25G01)*.

Every GQL-transaction is terminated by an attempt to either commit or a rollback. A successful rollback causes the transaction to have no effect on the GQL-catalog or GQL-data; a successful commit causes the execution outcome to be completely successful.

The GQL-agent that initiated a transaction can request the GQL-implementation to either commit or rollback that transaction by submitting a GQL-request that specifies a transaction commit command or a transaction rollback command to the GQL-server.

The transaction demarcation commands may be submitted in the same GQL-request as a procedure. A GQL-request may state a <start transaction command> prior to the procedure, allowing specifying characteristics of the transaction started for that procedure. A GQL-request may state a transaction commit command or a transaction rollback command following the procedure. A GQL-request containing a transaction commit command following a procedure causes a request to the GQL-server to terminate the currently active GQL-transaction by committing upon the successful completion of that procedure.

A GQL-request containing a transaction rollback command following a procedure causes a request to the GQL-server to terminate the currently active GQL-transaction by rolling back upon the successful completion of that procedure.

If a GQL-transaction becomes blocked, cannot complete without causing semantic inconsistency, or if the resources required to continue its execution become unavailable, then it is implementation-dependent (UA007) whether or not a rollback is forced.

Any execution of a procedure with a failed outcome will cause an attempt by the GQL-implementation to rollback the current transaction.

Once a commit or rollback has been issued, then no further procedures will be executed in the sequence of procedures that make up the transaction.

Once requested, transaction termination either succeeds or fails. If a termination request cannot be processed successfully, then the state of the GQL-catalog and GQL-data becomes indeterminate. The ways in which termination success or failure statuses are made available to the GQL-agent or to an administrator is implementation-defined (IW005).

Starting and terminating of transactions may be accomplished by implementation-defined (IW004) means (including proprietary agent-server protocols, transaction managers that act out of band to agent-server interactions, auto-starting, and auto-rollback on exception of transactions). The behavior of transactions started and terminated by an implementation-defined means shall be the same as transactions started with a <start transaction command> and terminated with an <end transaction command>.

4.6.3 Transaction isolation

Provisional (uncommitted) changes to the GQL-catalog or GQL-data state that are made in the context of a transaction executing a catalog-modifying procedure or a data-modifying procedure may be visible, at some point or to some degree, to other GQL-agents executing a concurrent transaction. Other than serializable, the levels of transaction isolation, their interactions, their granularity of application, and the means of selecting them are implementation-defined (IE002).

NOTE 34 — GQL-implementations are therefore free to use the isolation levels defined in SQL, or more recently-defined levels such as Snapshot Isolation or Serializable Snapshot Isolation, or other industrially-applied or theoretical variants.

4.6.4 Encompassing transaction belonging to an external agent

In some environments (e.g. remote database access), a GQL-transaction can be part of an encompassing transaction that is controlled by an agent other than the GQL-agent.

In such environments, an encompassing transaction is terminated via that other agent, which interacts with the GQL-implementation via an interface that may be different from GQL (COMMIT or ROLLBACK), in order to coordinate the orderly termination of the encompassing transaction. If the encompassed GQL-transaction is terminated by an implicitly initiated <rollback command>, then the GQL-implementation will interact with that other agent to terminate that encompassing transaction. The specification of the interface between such agents and the GQL-implementation is beyond the scope of this document. However, it is important to note that the semantics of a GQL-transaction remain as defined in the following sense:

- When an agent other than the GQL-agent requests the GQL-implementation to rollback a GQL-transaction, the General Rules of Subclause 8.3, "<rollback command>", are performed.
- When such an agent requests the GQL-implementation to commit a GQL-transaction, the General Rules of Subclause 8.4, "<commit command>", are performed. To guarantee orderly termination of the encompassing transaction, this commit operation may be processed in several phases not visible to the application; it is not required that all of the General Rules of Subclause 8.4, "<commit command>", are executed in a single phase.

4.7 GQL-requests and GQL-programs

4.7.1 General description of GQL-requests and GQL-programs

A GQL-request is the basic unit of communication between the GQL-client and the GQL-server.

A GQL-request is described by a GQL-request descriptor that comprises:

- The GQL-program, which is a *GQL source text*.

A GQL source text is a character string whose character repertoire shall be an implementation-defined (IV001) subset of the Universal Character Set repertoire specified by The Unicode® Standard that includes every <GQL language character> and every <other language character> supported by the GQL-implementation.

NOTE 35 — Any additional character not included in the character repertoire of GQL source text can still be represented in character string literals and identifiers by an <escaped character>.

- The GQL-request parameters, which are represented as a (possibly empty) dictionary of general parameters that is implicitly determined using an implementation-defined (IW006) mechanism.

If the GQL-program is GQL source text in conforming GQL language, then it is generated by a <GQL-program> *PROG* that specifies the commands and GQL-procedures that are to be executed; otherwise, an exception condition is raised as specified by Subclause 4.7.3, “Execution of GQL-requests”. The request outcome of a GQL-request is the execution outcome of the requested GQL-program.

4.7.2 GQL-request contexts

4.7.2.1 Introduction to GQL-request contexts

A GQL-request context is a context that augments a session context in which an individual GQL-request is executed and that comprises the following characteristics:

- The dynamic parameters, which are represented as a (possibly empty) dictionary of general parameters.
- The request timestamp, which is either “not set” or a zoned datetime at which every <datetime value function> in the GQL-program is effectively evaluated.

NOTE 36 — See Subclause 4.16.6, “Temporal types”.

- The execution stack, which is a push-down stack of execution contexts.
- The request outcome, which is an execution outcome.

In this document the following phrases are used:

- The *current dynamic parameters* are the dynamic parameters of the current request context.
- The *current request timestamp* is the request timestamp of the current request context.
- The *current execution stack* is the execution stack of the current request context.
- The *current request outcome* is the request outcome of the current request context.

4.7.2.2 GQL-request context creation

A new GQL-request context *RCX* is created and initialized with a dictionary of dynamic parameters *DYN_PARAMS* as follows:

NOTE 37 — The provisions specified in Subclause 4.7.3, “Execution of GQL-requests” define the current request context of the currently executing GQL-request.

- The dynamic parameters of *RCX* are set to *DYN_PARAMS*.
- The request timestamp of *RCX* is set to “not set”.
- The execution stack of *RCX* is set to a new empty execution stack.
- The request outcome of *RCX* is set to a successful outcome with an omitted result.

4.7.2.3 GQL-request context modification

In this document, application of General Rules only modifies an already initialized GQL-request context by:

- Pushing a new execution context onto its execution stack.
- Popping an execution context from its execution stack.
- Setting its request outcome.
- Setting its request timestamp.

4.7.3 Execution of GQL-requests

If a GQL-request *REQUEST* is submitted by the GQL-client on behalf of the GQL-agent to the GQL-server in the context of the GQL-environment, then *REQUEST* is executed by effectively performing the following steps:

- 1) Let *SESSION* be the GQL-session in which *REQUEST* was submitted. If no session context has been associated with *SESSION*, then a new session context is created and initialized, as specified in Subclause 4.5.2.2, “Session context creation”, and associated with *SESSION*.
NOTE 38 — This ensures that the current session context is defined.
- 2) Let *SCX* be the current session context and let *SES_SCHEMA*, *SES_GRAPH* and *SES_PARAMS* be the session schema, the session graph, and the session parameters, respectively, of *SCX*.
- 3) Let *DYN_PARAMS* be the dictionary of general parameters comprising every session parameter of *SCX* whose parameter name is not the parameter name of a GQL-request parameter of *REQUEST* and every GQL-request parameter of *REQUEST*.
- 4) Let *PROG* be determined by effectively performing the following steps:
 - a) If the GQL-program *SOURCE* of *REQUEST* conforms to the Format of <GQL-program>, then *SOURCE* forms the BNF non-terminal instance *PROG* that is determined by the left normal form derivation of *SOURCE* from <GQL-program>; otherwise, an exception condition is raised: *syntax error or access rule violation (42000)*.
 - b) The Syntax Rules of Subclause 22.1, “Annotation of a <GQL-program>”, are applied with *PROG* as *GQL_PROGRAM*, *SES_SCHEMA* as *INI_SCHEMA*, *SES_GRAPH* as *INI_GRAPH*, *SES_PARAMS* as *SES_PARAMS*, and *DYN_PARAMS* as *DYN_PARAMS*.
 - c) The Syntax Rules and Conformance Rules of Clause 6, “<GQL-program>” are applied to *PROG*.
 - d) The execution of *PROG* shall be permitted according to the provisions of Subclause 4.2.4.2, “Authorization identifiers and privileges”.
 - e) If performing any of the preceding steps fails, then an exception condition is raised. Unless a Syntax Rule, Access Rule, or other provision was violated in the determination of *PROG* that specifies an explicit exception condition, an exception condition is raised: *syntax error or access rule violation (42000)*.

ISO/IEC 39075:2024(en)
4.7 GQL-requests and GQL-programs

- 5) Let *RCX* be a new request context that is initialized with *DYN_PARAMS* as its dictionary of dynamic parameters.
- 6) The current request context is set to *RCX*.
- 7) Case:
 - a) If the preceding determination of *PROG* caused exception conditions to be raised, then the current request outcome is set to a failed outcome that records these exception conditions in accordance with the provisions of Subclause 4.9.3, “Conditions”. If multiple exception conditions with the same highest priority have been raised, then it is implementation-dependent (UW001) which of these exception conditions is set as the status of the current request outcome.
 - b) Otherwise, the General Rules of Clause 6, “<GQL-program>” are applied to *PROG* in a new root execution context. The provisions of Subclause 4.6, “GQL-transactions”, are effective. If this application of General Rules fails and thus causes the GQL-implementation to rollback the currently active GQL-transaction, then the current transaction is set to “not set”.

NOTE 39 — In the case of failure, the current execution stack is effectively abandoned.
- 8) If the current transaction is active and the current termination flag is set to *True*, then the General Rules of Subclause 8.3, “<rollback command>”, are applied in a new root execution context.

NOTE 40 — This sets the current request outcome to a failed outcome.
- 9) The current request outcome is returned to the GQL-client for delivery to the GQL-agent.

NOTE 41 — This includes the result (if any), the declared type of the result (if available), and all recorded diagnostic information. See Subclause 4.8.4, “Execution outcomes”.
- 10) If the current termination flag is set to *True*, then *SCX* is dissociated from *SESSION* and *SCX* and *RCX* may be destroyed; otherwise, the current termination flag is set to *False* and the current request context is set to “not set”.

4.7.4 Working schema references

In addition to the current request context and the current session context, the execution of procedures, statements, and commands interacts with GQL-schemas identified by working schema references.

A *working schema reference* is an <absolute catalog schema reference> identified by the Syntax Rules of a BNF non-terminal instance, such as a <GQL-program> or a <use graph clause>. A working schema reference has a scope. A BNF non-terminal instance that identifies a working schema reference has a scope clause.

In this document, the phrase *current working schema reference* of *A* is used to refer to one specific working schema reference of a BNF non-terminal instance *A* (unless “omitted”), defined as follows.

Case:

- 1) If there is a working schema reference *WSR* identified by the BNF non-terminal instance with the innermost scope clause containing *A* and the scope of *WSR* includes *A*, then the current working schema reference of *A* is *WSR*.
- 2) Otherwise, the current working schema reference of *A* is “omitted”.

The phrase *current working schema* of *A* is used to refer to one specific GQL-schema (unless “omitted”), defined as follows.

Case:

- 1) If the current working schema reference of *A* is defined, then the current working schema of *A* is the GQL-schema identified by the current working schema reference of *A*.

ISO/IEC 39075:2024(en)
4.7 GQL-requests and GQL-programs

NOTE 42 — The GQL-schema identified by an <absolute catalog schema reference> is defined by the Syntax Rules of [Subclause 17.1](#), “<schema reference> and <catalog schema parent and name>”.

2) Otherwise, the current working schema of *A* is “omitted”.

4.7.5 Working graph site

In addition to the current request context, the current session context, and working schema references, the execution of procedures, statements, and commands interacts with graphs that are the referent of graph reference values occupying sites identified as working graph sites.

A working graph site is a site whose declared type is a graph reference value type and that is identified by the Syntax Rules of a BNF non-terminal instance, such as a <GQL-program> or a <use graph clause>. A working graph site has a scope. A BNF non-terminal instance that identifies a working graph site has a scope clause. Once a working graph site *WGS* is occupied, the graph referenced by *WGS* is the graph that is the referent of the graph reference value occupying *WGS*.

NOTE 43 — In this document, the phrase “graph referenced by a working graph site” is not used in Syntax Rules.

In this document, the phrase *current working graph site* of *A* is used to refer to one specific working graph site of a BNF non-terminal instance *A* (unless “omitted”), defined as follows.

Case:

- If there is a site *WGS* that is identified as a working graph site by the BNF non-terminal instance with the innermost scope clause containing *A* and the scope of *WGS* includes *A*, then the current working graph site of *A* is *WGS*.
- Otherwise, the current working graph site of *A* is “omitted”.

4.7.6 Execution stack

The execution stack is a push-down stack of execution contexts associated with a GQL-request. The execution stack is created and initialized with a new execution context explicitly by the General Rules of [Clause 6](#), “<GQL-program>”. This newly created execution context is always the lowest execution context in the execution stack.

An execution context is pushed on the stack for each procedure or command that is executed, and is removed when that procedure or command completes execution. Child execution contexts may be pushed on the stack for the execution of a statement, the evaluation of an expression, or the processing of some other BNF non-terminal instance. If such an execution context has been created, it is removed when that execution, evaluation, or processing completes.

NOTE 44 — Changes to the execution stack are always explicitly specified in [Clause 4](#), “Concepts” or by General Rules.

The current execution context is always the highest execution context in the execution stack of the currently executing GQL-request.

4.7.7 Operations

[Subclause 3.5](#), “Procedure and command terms and definitions” defines an operation as an identifiable action carried out by an execution. Operations are either indivisible or comprise multiple successive suboperations.

The execution of a GQL-request is the operation that evaluates the specified GQL-program of that GQL-request.

NOTE 45 — See [Subclause 4.7.3](#), “Execution of GQL-requests”.

ISO/IEC 39075:2024(en)
4.7 GQL-requests and GQL-programs

The execution of a component of the GQL language specified by a BNF non-terminal instance is the operation that evaluates that BNF non-terminal instance. In particular:

- A GQL-program is executed by evaluating the specified GQL-procedure or the specified commands of the GQL-program.
- A GQL-procedure is executed by evaluating the variable definitions and statements of the GQL-procedure that are specified in the <procedure body>.
- An external procedure is executed by evaluating its procedural logic.
- A command is executed by evaluating the specified syntactic elements of the command.
- A statement is executed by evaluating the specified syntactic elements of the statement.

Certain operations can cause one or more side effects specified in this document. Such side effects can be session-modifying, transaction-modifying, catalog-modifying, or data-modifying.

NOTE 46 — See Subclause 3.5, “Procedure and command terms and definitions”.

The side effects of a composite operation include the side effects of its suboperations, if any.

The construction of a new GQL-object is not considered as causing a catalog-modifying side effect or a data-modifying side effect, per se. However the assignment of such a newly constructed GQL-object to certain sites is possibly considered as causing a catalog-modifying side effect or a data-modifying side effect, depending on the nature of the site.

NOTE 47 — As an example, the construction and initial population of a transient binding table that is not assigned to a site in the GQL-catalog or the session context during the application of General Rules is not considered a catalog-modifying side effect or a data-modifying side effect.

Operations that are performed as part of executing a GQL-request are considered as *successful operations* or as *failed operations*, respectively, depending on the current execution outcome present immediately after their completion.

4.8 Execution contexts

4.8.1 General description of execution contexts

An execution context is a context comprising objects that are associated with and manipulated by the execution of operations. It provides access to visible objects, allowing their manipulation by the execution of a procedure or a command.

An execution context comprises the following characteristics:

- The working record, which is a record.
- The working table, which is a binding table.
- The execution outcome, which is an execution outcome.

For every execution context *CONTEXT* created or modified by the General Rules specified in this document, it always holds that the working table and the working record of *CONTEXT* are field name-disjoint.

The atomicity of multiple operations that are executed in an execution context depends on the types of the operations.

The current execution context is the execution context of the currently executing operation of the currently executing procedure, command, statement, or evaluation of an expression.

In this document the following phrases are used:

- The *current working record* is the working record of the current execution context.

- The *current working table* is the working table of the current execution context.
- The current execution outcome is the execution outcome of the current execution context.
- The current execution result is the result of the current execution outcome.

In this document, the Syntax Rules applying to a BNF non-terminal instance *X* that is contained in a <GQL-program> can refer to the following additional transient sites:

- The *incoming working record* of *X* is the current working record immediately before the application of the first General Rule of *X*.
- The *incoming working table* of *X* is the current working table immediately before the application of the first General Rule of *X*.
- The *outgoing working record* of *X* is the current working record immediately after the application of the last applicable General Rule of *X*.
- The *outgoing working table* of *X* is the current working table immediately after the application of the last applicable General Rule of *X*.

If a BNF non-terminal instance *INNER* is immediately contained in a BNF non-terminal instance *OUTER* and if neither the declared type of the incoming working record of *INNER* nor the declared type of the incoming working table of *INNER* is specified explicitly, then all of the following conditions are implicit:

- The declared type of the incoming working record of *INNER* is the declared type of the incoming working record of *OUTER*.
- The declared type of the incoming working table of *INNER* is the declared type of the incoming working table of *OUTER*.

If every possible instance *OUTER* of a BNF non-terminal symbol <OUTER> immediately contains exactly one BNF non-terminal instance *INNER* and if neither the declared type of the outgoing working record of *OUTER*, nor the declared type of the outgoing working table of *OUTER*, nor the declared type of *OUTER* itself is specified explicitly, then all of the following conditions are implicit:

- For the declared type of the outgoing working record of *OUTER*:

Case:

- 1) If the declared type of the outgoing working record of *INNER* is not “omitted”, then the declared type of the outgoing working record of *OUTER* is the declared type of the outgoing working record of *INNER*.
- 2) Otherwise, the declared type of the outgoing working record of *OUTER* is “omitted”.

- For the declared type of the outgoing working table of *OUTER*:

Case:

- 1) If the declared type of the outgoing working table of *INNER* is not “omitted”, then the declared type of the outgoing working table of *OUTER* is the declared type of the outgoing working table of *INNER*.
- 2) Otherwise, the declared type of the outgoing working table of *OUTER* is “omitted”.

- For the declared type of *OUTER*:

Case:

- 1) If the declared type of *INNER* is not “omitted”, then the declared type of *OUTER* is the declared type of *INNER*.

- 2) Otherwise, the declared type of *OUTER* is “omitted”.

For every BNF non-terminal instance *NT* that has an execution outcome: if and only if *NT* has an omitted result, then the declared type of *NT* is the empty type.

NOTE 48 — See Subclause 4.16.8, “Immaterial value types: null type and empty type”.

4.8.2 Execution context creation and initialization

A new execution context is be created by one of the following approaches:

- Creating a new *root execution context RTX* that is initialized as follows:
 - The working table of *RTX* is set to a new unit binding table.
 - The working record of *RTX* is set to a new empty record.
 - The execution outcome of *RTX* is set to a successful outcome with an omitted result.
- Creating a new *child execution context CTX* that is initialized as follows:
 - The working record of *CTX* is set to a copy of the current working record.
 - The working table of *CTX* is set to a unit binding table.
 - The execution outcome of *CTX* is set to a copy of the current execution outcome.

The creation of a new child execution context may be further modified by overriding some of its characteristics *CHARACTERISTIC* using a phrase such as “with ... as its *CHARACTERISTIC*” or similar grammatical variants.

The execution of a *PROGRAM* in a new root execution context using a phrase such as “perform *PROGRAM* in a new root execution context” or similar grammatical variants is defined as follows:

- 1) A new root execution context *CTX0* is created.

NOTE 49 — This establishes the current execution context.
- 2) *CTX0* is pushed onto the current execution stack.
- 3) *PROGRAM* is performed.
- 4) The current request outcome is set to the current execution outcome.
- 5) The current execution stack is popped.

NOTE 50 — This empties the current execution stack.

NOTE 51 — A GQL-implementation can choose to implement an instruction to execute an *ACTION* in a new child execution context by executing it directly in the current execution context instead, as long as that direct execution is indistinguishable from the execution in a new child execution context that was specified by the instruction.

The execution of an *ACTION2* in a new child execution context amended with some record *R* using a phrase such as “perform *ACTION2* in a new child execution context amended with *R*” or similar grammatical variants is defined as follows:

- 1) Let *AWR* be the current working record amended with *R*.

NOTE 52 — Amending of records is defined in Subclause 4.15.4, “Record types”.
- 2) Let *ACTION1* be defined as the following sequence of actions:
 - a) Set the current working record to *AWR*.
 - b) Perform *ACTION2*.

- 3) Perform *ACTION1* in a new child execution context.

For a BNF non-terminal instance *NT*, the phrases *is the result of NT*, *of the result of NT*, *is the value of NT*, and *of the value of NT* or similar grammatical variants are used in General Rules to refer to the result of the execution outcome present after a single evaluation of *NT* in a new execution context that copies each of its characteristics from the corresponding characteristics of its parent execution context.

NOTE 53 — If these phrases are used for the same non-terminal instance in two separate current execution contexts, then these effectively represent two independent evaluations in two separate child execution contexts.

4.8.3 Execution context modification

In this document, the characteristics of an execution context are only modified after their initialization by:

- Setting the working table.
- Setting the working record.
- Setting the execution outcome.

4.8.4 Execution outcomes

An execution outcome is a component of an execution context representing the outcome of an execution and comprises:

- A *status*, which is the (primary) GQL-status object of the execution outcome.
- A (possibly empty) list of additional GQL-status objects. This list is empty by default unless specified otherwise.
- An optional *result*, which (if present) is either a regular result or an omitted result (as explained below).

An execution outcome is one of:

- A successful outcome, which is an execution outcome recording a result. The result of a successful outcome is one of:
 - A regular result is a result that is a value or a binding table produced by the successful execution of an operation.
 - An omitted result is a result indicating the successful execution of an operation that produced no value.
- A failed outcome, which is an execution outcome recording an exception condition. The failed outcome has no result.

In particular, the result of a successful outcome can be any of the following:

- The result of a successful outcome representing the outcome of an execution of a procedure, a command, a statement, or an expression that is a <composite query expression>, or a BNF non-terminal instance related to the execution of one of these items is either an omitted result or a binding table.
- The result of a successful outcome representing the outcome of an execution of an expression that is a <value expression> is a value.
- The result of a successful outcome representing the outcome of an execution of an expression that is a <graph expression> is a graph reference value.

- The result of a successful outcome representing the outcome of an execution of an expression that is a <binding table expression> is a binding table reference value.

NOTE 54 — As a counterexample, the <use graph clause> and the <at schema clause> are not executed. They do not have any General Rules and do not interact with an execution context. Consequently, they have neither an execution outcome nor a result.

It is implementation-defined (IA001), whether the declared type of a regular result of a successful outcome of a GQL-request is exposed to the GQL-client. The empty type that is the declared type of an omitted result of a successful outcome of a GQL-request is not exposed to the GQL-client.

The GQL-status object of a successful outcome has a GQLSTATUS that corresponds to a completion condition and that indicates that any previous errors have been recovered and no new exception condition was raised by the last operation executed in the execution context.

The GQL-status object of a failed outcome has a GQLSTATUS that corresponds to an exception condition and that indicates that a previously raised exception condition was not recovered or that a new exception condition was raised by the last operation executed in the execution context.

If a non-terminal *NT* has an outcome, then it is an execution outcome and the result and status of *NT* are the result and status, respectively, of the outcome of *NT*.

4.9 Diagnostic information

4.9.1 Introduction to diagnostic information

Following the execution of a GQL-request, diagnostic information is returned to the GQL-client that originated that GQL-request.

This diagnostic information is contained in the status of an execution outcome that describes the returned (primary) condition and that may also include additional implementation-defined (ID017) diagnostic information.

NOTE 55 — See Subclause 4.8.4, “Execution outcomes”.

It is implementation-defined (IW007) how a GQL-status object is presented to a GQL-client.

NOTE 56 — For example, in a Java environment a GQL-status object with a GQLSTATUS whose corresponding condition’s category is “X” could cause a Java runtime exception to be thrown.

The diagnostic information made available to the GQL-client is the diagnostic information recorded in the current request outcome.

NOTE 57 — See Subclause 4.5.2, “Session contexts”.

4.9.2 GQL-status objects

A *GQL-status object* represents a *condition* together with all related diagnostic information.

NOTE 58 — Conditions are introduced in Subclause 4.9.3, “Conditions”.

Whenever a GQL-request is executed, it sets values, representing one or more conditions resulting from that execution, in a GQL-status object. These values give some indication of what has happened. The diagnostic information is returned as part of the request outcome to the GQL-client.

Each GQL-status object comprises:

- A GQLSTATUS, which is a character string identifying a condition as defined in Subclause 23.1, “GQLSTATUS”.
- A *status description*, which is a character string describing the GQLSTATUS.
- A record with diagnostic information as defined in Subclause 23.2, “Diagnostic records”.

ISO/IEC 39075:2024(en)
4.9 Diagnostic information

- An optional nested GQL-status object for providing additional diagnostic information, such as a cause.

The diagnostic record can include additional implementation-defined (ID017) fields. The names of such fields shall start with underscore ('_').

Without Feature GA08, “GQL-status objects with diagnostic records”, a GQL-status object in a conforming GQL-implementation shall not contain a diagnostic record.

A status description is the concatenation of either a standard description or an implementation-defined (ID016) translation of that standard description appropriate for the locale of the GQL-client, and any implementation-defined (IV016) additional text about the condition provided by the GQL-implementation.

A *standard description* is defined as follows:

- 1) Let *CAT*, *SUBCLASS*, *COND*, and *SUBCOND* be the Category, Subclass, Condition, and Subcondition fields of the associated row in Table 8, “GQLSTATUS class and subclass codes”, for the condition corresponding to the GQLSTATUS.
- 2) Let *TEXT* be defined as follows.
Case:
 - a) If *CAT* is 'S', 'N', or 'X', then
Case:
 - i) If *SUBCLASS* is '000', then *TEXT* is *COND*.
 - ii) Otherwise, *TEXT* is the concatenation of *COND*, ' - ', and *SUBCOND*.
 - b) Otherwise, *CAT* is 'W' or 'I', and *TEXT* is *SUBCOND*.
- 3) Case:
 - a) If *CAT* is 'X', then the standard description is the concatenation of 'error: ' and *TEXT*.
 - b) If *CAT* is 'W', then the standard description is the concatenation of 'warn: ' and *SUBCOND*.
 - c) If *CAT* is 'S' or 'N', then the standard description is the concatenation of 'note: ' and *TEXT*.
 - d) Otherwise, *CAT* is 'I' and the standard description is the concatenation of 'info: ' and *TEXT*.

At the beginning of an execution, the current GQL-status object is emptied. A GQL-implementation places diagnostic information about a completion condition or an exception condition corresponding to the GQLSTATUS into this GQL-status object. If other conditions are raised, the extent to which further GQL-status objects are included is implementation-defined (IA002).

4.9.3 Conditions

A condition is represented by the GQLSTATUS character string to which it corresponds. The format of GQLSTATUS character strings is specified in Subclause 23.1, “GQLSTATUS”.

There are two types of conditions:

- completion conditions.
- exception conditions.

A *completion condition* is one that permits the execution of a GQL-request to have an effect other than that associated with raising the condition. The completion conditions comprise the conditions: *successful completion* (00000), *warning* (01000), and *no data* (02000), including all subclass code variants.

ISO/IEC 39075:2024(en)
4.9 Diagnostic information

The completion condition *warning* (01000) is broadly defined as completion in which the effects are correct, but there is reason to caution the user about those effects. The subclass code provides information as to the specific reason for the warning. It is raised for implementation-defined (IE008) conditions as well as conditions specified in this document. The completion condition *no data* (02000) has special significance and is used to indicate an empty result (e.g., an empty tabular result). The completion conditions *successful completion* (00000), including any implementation-defined subclasses, as well as the completion condition *successful completion — omitted result* (00001) are defined to indicate a completion condition that does not correspond to *warning* (01000) or *no data* (02000).

If no other completion or exception condition has been specified, then the completion condition *successful completion* (00000) is returned. This includes conditions represented by a GQLSTATUS whose subclass code provides implementation-defined (IE010) information of a non-cautionary nature.

A successful outcome with a regular result returns the completion condition *successful completion* (00000), including any implementation-defined subclasses.

A successful outcome with an omitted result returns the completion condition *successful completion — omitted result* (00001).

An *exception condition* is one that causes the execution of a GQL-request to have no effect other than that associated with raising the condition (that is, not a completion condition).

The informational conditions, *informational* (03000), are secondary conditions that provide additional information that is of interest to the user and are only returned in the list of additional GQL-status objects of an execution outcome. The subclass code provides information as to the specific reason for the informational condition. The implementation-defined (IE009) informational conditions shall never be returned as the status of an execution outcome.

Except where otherwise specified, the phrase “an exception condition is raised:”, followed by the name of a condition, is used in General Rules and elsewhere in this document to indicate that:

- The execution of the GQL-request is unsuccessful.
- The application of the General Rules is terminated, unless explicitly stated otherwise.
- Diagnostic information is recorded in the current request outcome.

NOTE 59 — The current request outcome is made available to the GQL-client by the provisions of Subclause 4.7.2, “GQL-request contexts”.

- Execution of the GQL-request is to have no effect on GQL-data or the GQL-catalog.

The phrase “a completion condition is raised:”, followed by the name of a condition, is used in General Rules and elsewhere in this document to indicate that application of General Rules is not terminated and diagnostic information is made available; unless an exception condition is also raised, the execution of the GQL-request is successful.

If more than one condition could have occurred as a result of the execution of a GQL-request, then it is implementation-dependent (UA002) whether diagnostic information pertaining to more than one condition is made available.

NOTE 60 — There are several circumstances in which a GQL-implementation does not necessarily detect all possible exception conditions. For example, during left to right evaluation of an expression when the result can be determined without evaluating the entire expression (See Subclause 5.3.2.4, “Rule evaluation order”, or during evaluation of a selective <path pattern> when not all candidate path bindings need to be tested (See NOTE 392 in Subclause 22.4, “Evaluation of a selective <path pattern>”). In these situations multiple conditions could be, but not necessarily are, detected. It is possible to make such multiple conditions available, but the exact circumstances in which this is done remains implementation-dependent (UA002).

Those additional conditions, if any, are placed in separate GQL-status objects in the list of additional GQL-status objects of the current request outcome.

For the purpose of choosing the status to be returned in an execution outcome:

- Every exception condition for transaction rollback has precedence over every other exception condition.
- Every exception condition has precedence over every completion condition.
- The completion condition *no data* (02000) has precedence over the completion condition *warning* (01000).
- The completion condition *warning* (01000) has precedence over the completion condition *successful completion* (00000) for any successful completion subclass.

4.10 Procedures and commands

4.10.1 General description of procedures and commands

Procedures and commands are executed as part of executing a GQL-program in order to:

- Read or modify the GQL-catalog and the catalog objects it contains.
- Read or modify GQL-data.
- Read or modify characteristics of the current session context (e.g., the session schema, the session graph, the session parameters, or the optional transaction).
- Read characteristics of the current request context (e.g., the current dynamic parameters or the current request timestamp).
- Read, modify, or manipulate other GQL-objects that are reachable via the current execution context, the current request context, or the current session context.

4.10.2 Procedures

4.10.2.1 General description of procedures

Subclause 3.5, “Procedure and command terms and definitions” defines a procedure as a description of a computation on input arguments whose execution computes an execution outcome and optionally causes side effects. In this document, every procedure specifies its *procedure logic* that comprises operations (such as statements) together with the order in which they have to be effectively performed to completely execute the computation described by the procedure.

These operations may read, modify, or otherwise manipulate the GQL-catalog, the GQL-data, the GQL-session, or other GQL-objects reachable via the current execution context, the current request context, or the current session context that are available during their execution.

A *named procedure* is a procedure that is defined in the GQL-catalog, i.e., it is a primary catalog object. An *inline procedure* is a procedure that is defined by a <procedure specification> in a conforming GQL-program.

In a <GQL-program>, a named procedure is specified by a <procedure reference> that identifies it while an inline procedure is specified directly by giving its definition.

A GQL-procedure is a procedure whose procedure logic is specified in the GQL language, specifically by a <procedure body>. An external procedure is a procedure provided via an implementation-defined (IW010) mechanism.

This document does not specify the mechanism by which a named procedure is defined in conforming GQL language. Consequently, a named procedure is an external procedure while an inline procedure is a GQL-procedure.

4.10.2.2 Named procedure descriptors

A named procedure is described by a named procedure descriptor that comprises:

- The list of the declared types of the mandatory procedure parameters and their names.
- The list of the declared types of the optional procedure parameters, their names, and their default values.

NOTE 61 — Optionality is only supported for a succession of parameters leading to the end of the list of parameters. If a named procedure call provides an argument for any one of a succession of optional parameters, then it must provide arguments for all preceding optional parameters. Comma-separated gaps in the argument list are not supported.

- The procedure result type.
- The indication of the kinds of side effects the execution of the named procedure can cause, specified as follows:
 - CATALOG PROCEDURE indicates that this procedure is a catalog-modifying procedure, as described in Subclause 4.10.2.4, “Procedures classified by kind of side effects”
 - DATA PROCEDURE indicates that this procedure is a data-modifying procedure, as described in Subclause 4.10.2.4, “Procedures classified by kind of side effects”.
 - QUERY PROCEDURE indicates that this procedure is a query procedure, as described in Subclause 4.10.2.4, “Procedures classified by kind of side effects”.

The parameter name of each mandatory and optional procedure parameter is unique within the named procedure descriptor.

For every named procedure *PROC*, the named procedure descriptor of *PROC* is a catalog object descriptor. The name of *PROC* (also known as procedure name) is the catalog object name of *PROC*.

For a given named procedure descriptor *PROCDESC*, let *LMPP* be the list of the declared types of the mandatory procedure parameters and their names that is included in *PROCDESC*. Let *LOPP* be the list of the declared types of the optional procedure parameters, their names, and their default values that is included in *PROCDESC*. The number of the procedure parameters required by *PROCDESC* is the number of elements of *LMPP*, the maximum number of the procedure parameters that are allowed by *PROCDESC* is the sum of the number of the procedure parameters required by *PROCDESC* and the number of elements of *LOPP*, and the list of procedure parameters of *PROCDESC* is the list formed by the elements of *LMPP* followed by the elements of *LOPP*.

4.10.2.3 Procedure execution

Procedures are executed by calling them in a specified execution context. In the case of named procedures, procedure call arguments are passed via the working record of that execution context. Prior to the execution of a named procedure call, the provided procedure call arguments shall fulfill the requirements of the named procedure descriptor of the called named procedure regarding their number and data type. Every optional parameter of the named procedure descriptor of the called named procedure that is not included in the provided procedure call arguments is defaulted as specified by the named procedure descriptor of the called named procedure.

The outcome of the execution *EXE* of a procedure is the execution outcome present in the specified execution context after execution has terminated. The result and status of *EXE* are the result and status, respectively, of the outcome of *EXE*.

4.10.2.4 Procedures classified by kind of side effects

Procedures are classified by the kind of side effects that their execution can cause. For this purpose, this document distinguishes between:

- A catalog-modifying procedure, which is a procedure whose execution can cause catalog-modifying side effects only or catalog-modifying side effects together with data-modifying side effects.
- A data-modifying procedure, which is a procedure whose execution can cause data-modifying side effects only.
- A query procedure, which is a procedure whose execution does not cause any side effects.

4.10.3 Commands

A command is an operation that is not specified by a GQL-procedure. The execution of a command computes an execution outcome and can cause side effects.

Every supported command is one of the following:

- A session command, i.e., a command that can only perform session-modifying side effects.
- A transaction command, i.e., a command that can only perform transaction-modifying side effects.

Commands are executed by calling them in a specified execution context.

NOTE 62 — All required arguments are supplied implicitly via the current execution context.

The outcome of the execution *EXE* of a command is the execution outcome present in the specified execution context after execution has terminated. The result and status of *EXE* are the result and status, respectively, of the outcome of *EXE*.

4.10.4 GQL-procedures

4.10.4.1 Introduction to GQL-procedures

A GQL-procedure is a procedure written in the GQL language, which is part of (or available to) a GQL-program.

The procedure logic of a GQL-procedure is specified by the <procedure body> provided when the procedure is defined.

A <procedure body> comprises:

- A (possibly empty) sequence of <binding variable definition>s that are specified by a <binding variable definition block> and that define fixed variables.
- A sequence of statements that are specified by the <statement block> directly contained in the <procedure body>.

The procedure logic of a GQL-procedure is executed in the current execution context by first executing its variable definitions followed by executing its statements. This is specified completely by [Subclause 9.2, “<procedure body>”](#).

This document does not specify the mechanism by which a procedure that is not a GQL-procedure is specified or provided to the GQL-server.

4.10.4.2 Binding variables and general parameters

GQL-procedures interact with the following kinds of variables and general parameters:

NOTE 63 — [Subclause 3.2, “GQL-environment terms and definitions”](#), defines a general parameter as a pair comprising a name and a pair comprising a value and a value type.

- One or more session parameters, where a session parameter is a general parameter that is defined in a session context.

- One or more dynamic parameters, where a dynamic parameter is a general parameter that is defined in a GQL-request context.
- One or more binding variables, where a binding variable is a variable bound to a value occupying a site that is assigned to the binding variable in the context of executing some operation. Every binding variable is one of the following (relative to a contextually implied execution context):
 - A *fixed variable* that is a binding variable assigned to the identified field of the working record.
NOTE 64 — A fixed variable is always bound to exactly one value when iterating over the current working table during procedure execution.
 - An *iterated variable* that is a binding variable assigned to the identified fields of the records of the working table.
NOTE 65 — An iterated variable *VAR* possibly binds to zero, one, or more values when iterating over a working table *TABLE*, e.g., during procedure execution. A concrete binding is realized through the creation of a child execution context *CONTEXT* for each iterated record *RECORD* from *TABLE* such that *VAR* is bound as a fixed variable in the working record of *CONTEXT* to the corresponding field value of *RECORD*.

Binding variables and parameters are defined in disjoint namespaces and syntactically disambiguated by always prefixing the parameter name specified by a <general parameter reference> with a <dollar sign> and the parameter name specified by a <substituted parameter reference> with a <double dollar sign>.

NOTE 66 — See Subclause 21.3, “<token>, <separator>, and <identifier>”.

4.10.4.3 Statements

A statement defines one or more operations executed as part of executing a GQL-procedure. Each operation updates the current execution context and its current execution outcome, and can cause side effects.

Many languages use a delimiter such as semicolon between the language’s statements. GQL does not use any delimiter between operations within a statement; statement components are identified by leading keywords.

4.10.4.4 Statements classified by use of working graph sites

A statement is classified according to its use of the <use graph clause>. Given a statement *S* and the result *S1* of applying all syntactic transformations to *S*, *S* is classified as follows:

- If neither *S* nor *S1* directly contain a <use graph clause>, then *S* is an *ambient statement*. An ambient statement *S* operates on the graph referenced by the current working graph site of *S*.
- If *S* or *S1* directly contains one or more <use graph clause>s, then *S* is a *focused statement*. A focused statement identifies one or more working graph sites and operates on the graphs referenced by these working graph sites.

4.10.4.5 Statements classified by function

A statement is classified according to its function.

- A *catalog-modifying statement* is a statement that can modify the catalog structure to create, and drop catalog objects and that can cause catalog-modifying side effects as well as optional additional data-modifying side effects to this end.
- A *data-modifying statement* is a statement that can perform insert, update, and delete operations on data objects and that can cause data-modifying side effects to this end.
- A *query statement* is a statement that can read persistent data and perform filtering, aggregating, and projecting operations.

4.11 Graph pattern matching

4.11.1 Summary of graph pattern matching

Graph pattern matching is the process of applying a list of special-purpose regular expressions called a *<graph pattern>* on a property graph *PG*, which in this document is always the current working graph, in order to return a set of reduced matches. Each reduced match is a list of path bindings; each path binding is a function that maps the symbols in a word of a regular language to a path of *PG*. The regular language and related concepts such as path binding and reduced match are specified in Subclause 22.2, “Machinery for graph pattern matching”, and Subclause 16.4, “*<graph pattern>*”.

A *<graph pattern>* is a list of *<path pattern>*s. Each *<path pattern>* in the list is matched to *PG* to detect a possibly-empty set of path bindings in *PG* that correspond to the *<path pattern>*.

The cross-product of these sets is reduced by natural joins over those global singleton element variables that are exposed by each *<path pattern>* and that are bound to the same graph element of *PG*. The remaining tuples are called reduced matches; the set of these reduced matches may be empty, but cannot be infinite because of syntactic restrictions to guard against infinite cycling.

The behavior of the graph pattern matching is defined in this document.

NOTE 67 — A more detailed summary can be found in Subclause 4.11.6, “Path pattern matching”.

Additional qualifying parameters (predicates, a selective *<path search prefix>*, a *<path mode>*, and the *<different edges match mode>*) that restrict the result may be supplied.

If *PG* contains cycles, then a match to a *<path pattern>* having an unbounded quantifier might return an infinite set of paths. However, this possibility is prevented by Syntax Rules that require the use of selective *<path search prefix>*s, restrictive *<path mode>*s, or the *<different edges match mode>* (or any combination thereof) to prevent infinitely-sized result sets.

4.11.2 Paths

A path *P* is a sequence of *N* graph elements of a property graph *PG*, such that:

- *N* is an odd number.
- If *N* = 1 (one), then *P* has no edges.
- The graph element at each odd index is a node and the graph element at each even index is an edge that connects the pair of nodes immediately before and after the edge in the sequence.

A path contains a non-empty sequence of nodes and a (possibly empty) sequence of edges. If there are two or more nodes, then the path is a sequence of graph elements that starts with a node and is followed by a sequence of ordered (edge, node) pairs.

If *PG* is a multigraph, then an edge in the path between a pair of nodes is one of possibly several edges between those nodes in *PG*.

Every edge *E* in the path has an orientation. If *E* is undirected, then the orientation of *E* is *undirected*. Let *V1* be the node that immediately precedes *E* in the path, and let *V2* be the node that immediately follows *E*. If the source of *E* is *V1* and the destination of *E* is *V2*, then the orientation of *E* is *left to right*. It is also said that the orientation of *E* is *directed pointing right*. If the source of *E* is *V2* and the destination of *E* is *V1*, then the orientation of *E* is *right to left*. It is also said that the orientation of *E* is *directed pointing left*.

NOTE 68 — A directed self-edge (i.e., when *E* is directed and *V1* and *V2* are the same node), is oriented both left to right and right to left.

If no edge from *PG* appears more than once in a path, then the path is called a *trail*. If no node from *PG* appears more than once in a path, except possibly as the first and last nodes of the path, then the path is called *simple*. If no node from *PG* appears more than once in a path, then the path is called *acyclic*.

NOTE 69 — The term “path” is used in more than one way in mathematical graph theory and in informal technical presentations and discussions. In this document the term path always denotes what a graph-theoretic work could call a partially-oriented walk in a pure property graph. Such a graph is a mixed multigraph; edges can be directed or undirected, and there can be multiple edges between two nodes.

4.11.3 Path patterns

A <path pattern> is an expression built from the following syntactic elements, governed by the Format and Syntax Rules of Subclause 16.4, “<graph pattern>”, and other Subclauses:

- An optional <path variable declaration>, to declare a path variable to be bound to a path binding.
- An optional <path pattern prefix>.

NOTE 70 — <path pattern prefix> is described in Subclause 4.11.7, “Path modes”, and Subclause 4.11.8, “Selective path search prefixes”, and specified in Subclause 16.6, “<path pattern prefix>”.

- A mandatory <path pattern expression>.

A <path pattern expression> is an expression built recursively from <element pattern>s, governed by the Format and Syntax Rules of Subclause 16.7, “<path pattern expression>”, and other Subclauses.

An <element pattern> is a pattern to match a single graph element. There are two kinds of <element pattern>s:

- <node pattern>:

A <node pattern> is a pattern to match a single node. A <node pattern> comprises at a minimum a matching pair of parentheses, which may contain optional <element pattern filler>, described subsequently.

- <edge pattern>:

An <edge pattern> is a pattern to match a single edge. An <edge pattern> is either a <full edge pattern> (which optionally permits <element pattern filler>) or an <abbreviated edge pattern> (which does not support <element pattern filler>). These two major classes of <edge pattern> have seven variants each, for the seven possible non-empty combinations of the three edge orientations (the individual edge orientations being directed pointing left, undirected, or directed pointing right). Thus there are fourteen varieties of <edge pattern>.

<element pattern filler> provides three optional components of <node pattern>s and <full edge pattern>s:

- <element variable declaration>, to declare an element variable to be bound to a graph element by the <element pattern>.
- <label expression>, a predicate regarding the labels of the graph element that is bound by the <element pattern>. A <label expression> is an expression formed from <label name>s and the <wildcard label> “%”, using the operation signs <vertical bar> “|” for disjunction, <ampersand> “&” for conjunction, <exclamation mark> “!” for negation, and balanced pairs of parentheses for grouping.
- <element pattern where clause>, a <search condition> to be satisfied by the graph element that is bound by the <element pattern>.

<path pattern expression>s are regular expressions built recursively from <element pattern>s using the following operations, governed by the Format and Syntax Rules of Subclause 16.7, “<path pattern expression>”, and other Subclauses.

- Concatenation, indicated syntactically by string concatenation (i.e., no operation sign).

ISO/IEC 39075:2024(en)
4.11 Graph pattern matching

NOTE 71 — <element pattern>s and more complex <path pattern expression>s can be concatenated in ways that appear to juxtapose either two <node pattern>s or two <edge pattern>s. These topologically inconsistent patterns are understood during pattern matching as follows:

- Two consecutive <node pattern>s bind to the same node.
 - Two consecutive <edge pattern>s conceptually have an implicit <node pattern> between them.
- Grouping, using matching pairs of parentheses to form a <parenthesized path pattern expression>. A <parenthesized path pattern expression> may optionally contain the following:
- A <subpath variable declaration> to declare a subpath variable.
 - A <search condition> to constrain matches.
- Alternation, indicated by <vertical bar> or <multiset alternation operator>.

NOTE 72 — Alternation with <vertical bar> provides set semantics using deduplication of redundant equivalent reduced matches, whereas alternation with <multiset alternation operator> provides multiset semantics, with no deduplication.

- Quantification, indicated by a postfix <graph pattern quantifier>, which may be affixed to an <edge pattern> or a <parenthesized path pattern expression>.
- <questioned path primary>, indicated by a postfix <question mark> affixed to an <edge pattern> or a <parenthesized path pattern expression>.

NOTE 73 — Unlike many regular expression languages, the <question mark> operator in <path pattern expression>s is not the same as {0,1}, the difference being that <questioned path primary> exposes its singleton element or subpath variables as conditional singletons, whereas {0, 1}, in common with all other quantifiers, exposes all element or subpath variables as group.

4.11.4 Graph pattern variables

A graph pattern variable *GPV* is a site identified by a <regular identifier> (the *name* of the graph pattern variable) and having a value determined by a multi-path binding *MPB* to a <graph pattern>.

There are four kinds of graph pattern variables:

- Node variables; the value of a node variable represents a list of bound nodes.
- Edge variables; the value of an edge variable represents a list of bound edges.
- Path variables; the value of a path variable represents a path binding.
- Subpath variables.

NOTE 74 — Subpath variables are not bound to a value in this document. They serve to assure multiset semantics in <path multiset alternation>.

In a <graph pattern binding table> *GPBT*, a <regular identifier> shall not identify more than one graph pattern variable; thus *GPBT* defines a namespace in which there is a one-to-one correspondence between the names of graph pattern variables and the graph pattern variables that they name.

NOTE 75 — Because of this one-to-one correspondence, certain terms that are defined for graph pattern variables are also defined for their names, so that the name of the variable and the variable itself can be used interchangeably in the rules. These terms include “declare” and “expose”.

Node variables and edge variables are collectively called element variables.

A node variable *NV* and its name are declared by an <element variable declaration> simply contained in a <node pattern>.

An edge variable *EV* and its name are declared by an <element variable declaration> simply contained in a <full edge pattern>.

An element variable may be declared in more than one <element pattern>. A multiply declared element variable *MDPV* expresses a natural equijoin in two circumstances:

- If *MDPV* is declared in both operands of a <path concatenation>.
- If *MDPV* is declared in two or more <path pattern>s of a <graph pattern>.

NOTE 76 — Declaring an element variable in two operands of a <path pattern union> or <path multiset alternation> does not express a natural join between the matches of the operands.

Element variables may be declared as temporary element variables. The scope of a temporary element variable is limited to the innermost <path term> that contains its declaration.

NOTE 77 — The introduction of new element variables by syntactic transformations possibly inadvertently changes the cardinality of results. In this document, this issue is avoided by generating syntax that declares such variables as temporary element variables. However, such syntax for the declaration of temporary element variables is not made available to the user.

A path variable *PV* and its name are declared by a <path variable declaration> simply contained in a <path pattern>. A path variable may only be declared once in a <graph pattern>.

A subpath variable *SV* and its name are declared by a <subpath variable declaration> simply contained in a <parenthesized path pattern expression>. More than one operand of a <path pattern union> may declare *SV*; otherwise, *SV* cannot be multiply declared in a <graph pattern>.

4.11.5 References to graph pattern variables

Graph pattern variables are visible within the <graph pattern binding table> *GPBT* in which they are declared. They can be referenced in <value expression>s, <search condition>s, and the <graph pattern yield clause> within *GPBT* where they are exposed as binding variables of the current working record and can be used accordingly. The binding table result of *GPBT* exposes yielded graph pattern variables as columns. Such a result is then consumed, e.g., by the containing <match statement> of *GPBT* by naturally joining it with the current working table to obtain a new current working table for further processing.

Assume a fixed <graph pattern binding table> as the implicit context for the remainder of this Subclause.

Element variables may be referenced in instances of the following BNF non-terminal symbols:

- <element pattern where clause>.
- <parenthesized path pattern where clause>.
- <graph pattern where clause>.
- <graph pattern yield clause>.

If an element variable *EV* is declared in a <quantified path primary> *QPP*, then it may bind to more than one graph element. References to *EV* are interpreted contextually. If the reference occurs outside *QPP*, then the reference is to the complete list of graph elements that are bound to *EV*. In this circumstance, the reference is said to have *group degree of reference*. If the reference does not cross a quantifier, then the reference has *singleton degree of reference* and references at most one graph element, even if the multi-path binding binds *EV* multiple times.

NOTE 78 — For example,

(X) -[E WHERE E.P > 1]->{1,10} (Y) WHERE SUM(E.P) < 100

This example references the edge variable *E* twice: once in the <edge pattern> and once outside the <edge pattern>. Within the <edge pattern>, *E* has singleton degree of reference and the <property reference> *E.P* references a property of a single edge. On the other hand, the reference within the SUM aggregate has group degree of reference (because of the quantifier {1, 10}) and references the list of edges that are bound to *E*.

A reference *R* to a graph pattern variable *GPV* is termed *local* in these circumstances:

ISO/IEC 39075:2024(en)
4.11 Graph pattern matching

- If *GPV* is declared in an <element pattern> *EP* and *R* is contained in the <element pattern where clause> of *EP*.
- If *GPV* is declared in a <parenthesized path pattern expression> *PPPE* and *R* is contained in the <parenthesized path pattern where clause> of *PPPE*.
- If *R* is in a <graph pattern where clause> or a <graph pattern yield clause>.

R has a degree of reference, determined by the Syntax Rules of Subclause 20.12, “<binding variable reference>”. The degree of reference of *R* is one of the following: unconditional singleton, conditional singleton, or effectively bounded group.

NOTE 79 — Subpath variables cannot be referenced by GQL language defined in this document, though it is possible they will be referenceable in a future version. At present their only use is to distinguish operands of a <path multiset alternation>.

NOTE 80 — In general, an element variable that is declared within a <quantified path primary> is bound by a multi-path binding to a list of graph elements. The reference *R*, on the other hand, can reference a proper subset of this list, based on the syntactic context in which *R* is placed. The degree of reference expresses the cardinality of the list that *R* references, as follows. An unconditional singleton references a list of cardinality 1 (one), a conditional singleton references a list of cardinality 0 (zero) or 1 (one), an effectively bounded group references a finite list. Syntax rules prohibit the possibility of referencing an infinite list.

A reference to a path variable always has unconditional singleton degree of reference.

References to graph pattern variables declared in *GPBT* are subject to the following constraints:

- An operand *OP* of <path pattern union> or <path multiset alternation> *U* may only reference element variables declared in *OP*, or outside of *U*.
- A non-local reference shall have singleton degree of reference.
- The group references in a <value expression> or a <dependent value expression> immediately contained in an <aggregate function> shall reference the same graph pattern variable.
- A selective <path pattern> *SPP* shall not reference a graph pattern variable that is not declared in *SPP*.

4.11.6 Path pattern matching

Path pattern matching is performed by Subclause 16.4, “<graph pattern>”, which in turn may invoke Subclause 22.3, “Evaluation of a <path pattern expression>”, and Subclause 22.4, “Evaluation of a selective <path pattern>”, as well as other Subclauses incidentally invoked for expression evaluation.

In more detail, a <path pattern> can be seen as a regular expression built from <node pattern>s and <edge pattern>s. To formalize this, an alphabet is formed (in Subclause 22.2, “Machinery for graph pattern matching”), comprising the element variable names, plus additional special symbols for the anonymous node symbol, the anonymous edge symbol, bracket symbols to indicate the beginning and ending of bindings to <parenthesized path pattern expression>s, and subpath symbols to mark the beginning and ending of subpaths. The precise specification of the corresponding regular language is found in Subclause 22.3, “Evaluation of a <path pattern expression>”.

Each <path pattern> of a <graph pattern> is evaluated independently of each other, resulting in a set of path bindings. A path binding is a list of elementary bindings; each elementary binding is an ordered pair (*LET*, *GE*), where *LET* is a member of the alphabet, and *GE* is

Case:

- if *LET* is a node variable or the anonymous node symbol, then a node;
- if *LET* is an edge variable or the anonymous edge symbol, then an edge;
- otherwise, equal to *LET*.

NOTE 81 — In the formal semantics, <label expression>s are evaluated at this stage, but <search condition>s are not; hence it is possible that some path bindings will be subsequently rejected because they fail a <search condition>. Implementations are of course free to “push down” predicate evaluation as long as the ultimate results are the same as prescribed by the formal semantics.

Projecting the elementary bindings of a path binding to the first component yields a word of the regular language of the <path pattern>. Projecting to the second component yields an annotated path, which is a path interspersed with mark-up to indicate the beginning and ending of <parenthesized path pattern expression>s and the beginning and ending of subpaths.

If a <path pattern> has an unbounded quantifier that is not in the scope of a restrictive <path mode> or the <different edges match mode>, then there may be infinitely many path bindings. Such a <path pattern> must have a selective <path search prefix> *SPSP*. Subclause 22.4, “Evaluation of a selective <path pattern>”, is invoked to reduce this potentially infinite set of path bindings to a finite set. All <search condition>s contained in the selective <path pattern> are evaluated to reduce the set of path bindings prior to making the final selection according to *SPSP*.

NOTE 82 — An implementation cannot generate an infinite set and then apply <search condition>s; instead it must enumerate the search space in a fashion enabling it to arrive at the same result as specified by the formal semantics. The formal semantics do not specify the algorithm for this enumeration, only the result.

At this point, there is a finite set of path bindings for each <path pattern> in a <graph pattern>. The cross product of these sets is formed; a member of the cross product is called a multi-path binding. A multi-path binding does not violate any restrictive <path mode> or <different edges match mode> that may be in force. The set of multi-path bindings is reduced by enforcing natural equijoins on the unconditional singleton variables exposed by the <path pattern>s.

Then the <search condition>s in the <graph pattern> are evaluated, potentially further reducing the set of multi-path bindings. Those that remain satisfy all the selective <path search prefix>es and all the <search condition>s of the <graph pattern>.

Then the function *REDUCE* (defined in Subclause 22.2, “Machinery for graph pattern matching”) is applied to the remaining multi-path bindings. *REDUCE* removes the elementary bindings of the bracket symbols, erases temporary element bindings by replacing them with corresponding anonymous node and edge symbol bindings, and collapses adjacent anonymous node symbol bindings into a single elementary binding. The results, now called reduced matches, are deduplicated according to set semantics.

NOTE 83 — If there is a <path pattern union> in the <graph pattern>, then duplicates can arise.

4.11.7 Path modes

A <path mode> may be specified for any <parenthesized path pattern expression> or any <path pattern> from the following choices:

- WALK, the default <path mode>, is the absence of any filtering implied by the other <path mode>s.
- TRAIL, where path bindings with repeated edges are not returned.
- ACYCLIC, where path bindings with repeated nodes are not returned.
- SIMPLE, where path bindings with repeated nodes are not returned unless these repeated nodes are the first and the last in the path.

Using trail, acyclic, or simple matching path modes for all unbounded quantifiers guarantees that the result set of a graph pattern matching will be finite.

NOTE 84 — There is no implied hierarchy between path modes. Specifically, path mode SIMPLE does not imply path mode TRAIL. While such an implication holds for directed graphs, it does not hold for graphs with undirected edges. For instance, the pattern $() - () - ()$ on a graph with two nodes and one undirected edge between them returns a match under SIMPLE but does not return any matches under TRAIL.

4.11.8 Selective path search prefixes

The set of path bindings resulting from a graph pattern match can be further restricted by a selective <path search prefix> *SPSP*. *SPSP* is defined by partitioning the potentially infinite set of path bindings by the endpoints, which are the first and last nodes bound by the path bindings.

NOTE 85 — This partitioning is crucial to the definition of *SPSP*. *SPSP* makes a selection of some number of path bindings from each partition. For example, a path binding is “shortest” if its length is minimal within its partition. A “shortest” path binding in one partition can be longer than a “shortest” path binding in another partition.

SPSP can constrain the result set in the following ways:

- <any path search>: to non-deterministically pick some number of path bindings from each partition; the number is specified by a <non-negative integer specification>.
- <all shortest path search>: to pick all the shortest path bindings in each partition.
- <counted shortest path search>: to non-deterministically pick some number of shortest path bindings from each partition; the number is specified by a <non-negative integer specification>.
- <counted shortest group search>: to group each partition into groups of path bindings having the same length, order the groups by path length, and pick all path bindings in some number of groups from the front of each partition; the number is specified by a <non-negative integer specification>.

The specification of *SPSP* guarantees that the result set of a graph pattern matching will be finite.

4.11.9 Match modes

A <graph pattern> *GP* may optionally specify a <match mode> that applies to all <path pattern>s simply contained in *GP*.

There are two <match mode>s:

- DIFFERENT EDGES: A matched edge is not permitted to bind to more than one edge variable. There are no restrictions on matched nodes.
- REPEATABLE ELEMENTS: There are no restrictions on matched edges or matched nodes.

If a <match mode> is not specified, then an implementation-defined (ID086) <match mode> is implicit.

4.12 Data types

4.12.1 General introduction to data types and base types

A data type is a set of elements with shared characteristics representing data. If *X* is an element of some data type *DT*, then *X* is said to be “of data type *DT*”. A data type characterizes the sites that may be occupied by instances of its elements. Every instance at a site belongs to one or more data types. The physical representation of an instance of a data type is implementation-dependent (UV005).

NOTE 86 — See Subclause 4.17, “Sites”.

A data type *A* can be equivalent to, disjoint from, overlap with, or be included in some other data type *B* depending on the sharing of elements between *A* and *B*. In this document, any two data types with the exact same elements are equivalent. All data types form a partial order that is implicitly defined by the inclusion relation between them. A data type containing all elements of some data type *X* is a supertype of *X*. A data type comprising elements contained in some data type *X* is a subtype of *X*. For a site *X* whose declared type is *DT* and a specification of data types *SODTS*, the phrase “*X* is of at least *SODTS*” expresses the condition of whether *DT* is a supertype of the data types specified by *SODTS*.

NOTE 87 — The system of data types defined in this document is structural. Values are not distinguished by tagging them with their type. However, the static base type and the most specific static value type of every

value are well-defined, as explained in [Subclause 4.12.7, “Open and closed data types”](#). A GQL-implementation could extend the type system with nominal types, e.g., by introducing a form of type name-tagged values. The details of such a hypothetical extension with nominal types are not considered in this document.

A base type is a set of data types with shared characteristics (i.e., it is a “type of types”). If T is a data type included in some base type BT , then T is said to be “of base type BT ”. The supertype and subtype relations on base types are defined analogous to the corresponding definitions for data types given above. Elements of a data type of some base type A may still be assigned to a site whose declared type is another data type of some other base type B , which is possibly different from A using relevant type conversions.

NOTE 88 — See [Subclause 4.17.3, “Assignment and store assignment”](#).

Every data type is associated with exactly one of its base types, which is called the *primary base type* of the data type. Primary base types indirectly determine the universe of elements from which each of their data types are drawn (e.g., the character strings). If a base type only includes data types whose primary base type is the base type itself, then the base type is a *static base type*; otherwise, the base type is a *dynamic base type*. Similarly, the data types whose primary base type is a static base type or a dynamic base type, are called *static data types* or *dynamic data types*, respectively.

Every data type defined in this document is included in its primary base type and every value type defined in this document is included in the dynamic base type of all value types (ANY DATA). For every dynamic union type, those two base types are identical.

NOTE 89 — See [Subclause 4.14, “Dynamic union types”](#).

4.12.2 Major classes of data types

[Figure 3, “Major classes of data types”](#) shows a pictorial overview of the major classes of data types defined in this document together with their base types and examples of data types of those base types. Every data type is either a GQL-object type or a value type. Additionally, every data type is either a material data type, a nullable data type, or an immaterial data type.

NOTE 90 — See [Subclause 4.12.5, “Material, nullable, and immaterial data types”](#).

ISO/IEC 39075:2024(en)

4.12 Data types

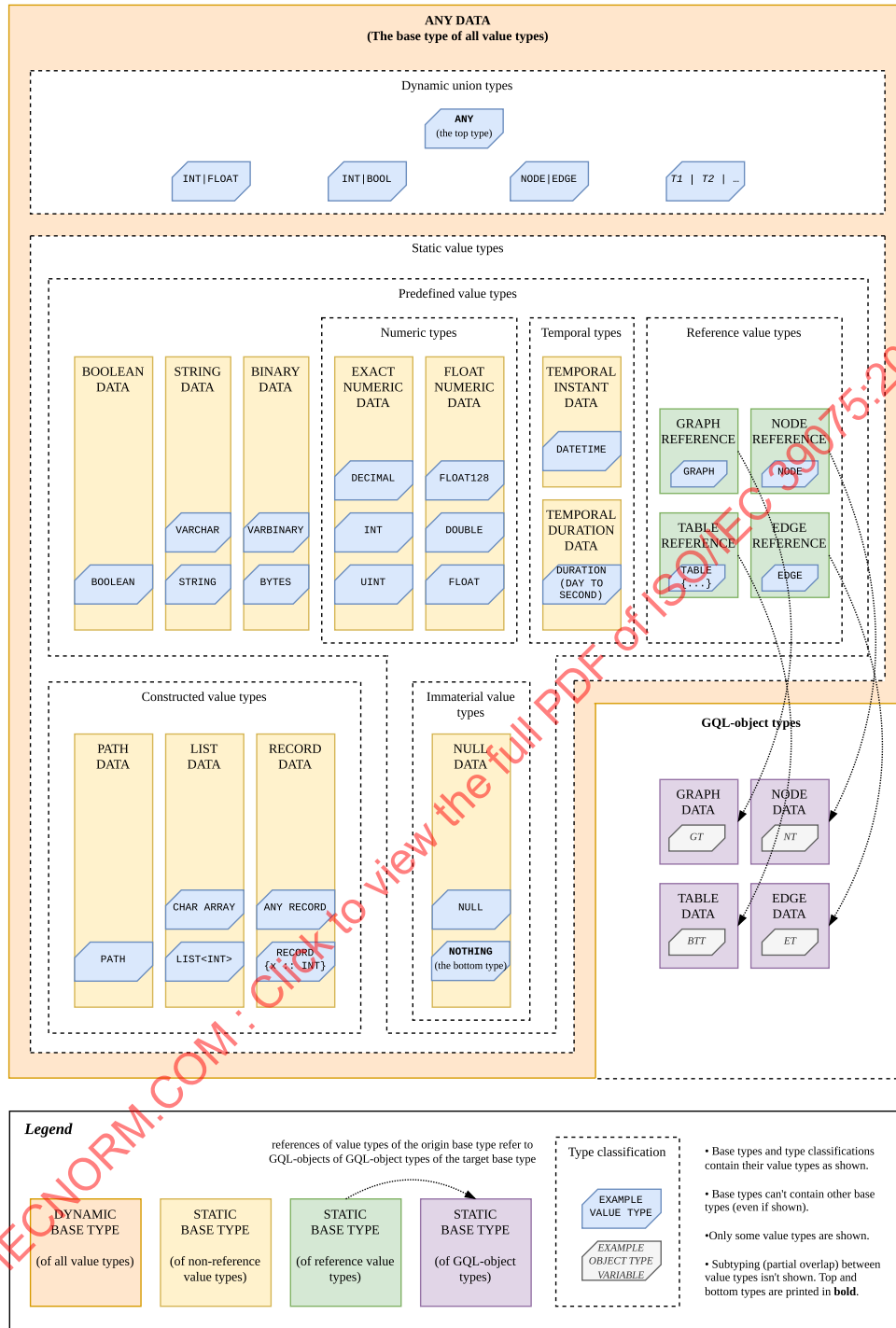


Figure 3 — Major classes of data types

A GQL-object type is a data type comprising GQL-objects, and a value type is a data type comprising values. Every GQL-object type defined in this document comprises either only primary objects or only secondary objects.

NOTE 91 — This document defines certain GQL-objects that do not have a corresponding (constraining) GQL-object type that contains them. See Subclause 4.3, “GQL-objects”.

NOTE 92 — The property values of GQL-objects are required to be included in certain value types. See Subclause 4.3.4, “Properties and supported property value types”.

NOTE 93 — The property types of graph element types and the column types of binding table types are not considered data types.

NOTE 94 — See Subclause 4.13, “GQL-object types”.

Every regular (non-immaterial) value type defined in this document is included in exactly one of the following classes of value types:

- A dynamic union type is a supertype of two or more static value types.
NOTE 95 — See Subclause 4.14, “Dynamic union types”.
- A constructed value type is a data type comprising composite elements (such as a path value type, a list value type, or a record type).
NOTE 96 — See Subclause 4.15, “Constructed value types”.
NOTE 97 — The field types of record types are not considered data types.
- A predefined value type is a value type specified in this document that is atomic and provided by the GQL-implementation. A value type is predefined even though the user is required (or allowed) to provide certain parameters when specifying it (e.g., the precision of a number). Predefined value types are sometimes called “built-in data types”, though not in this document. A reference value type is a predefined value type comprising reference values.
NOTE 98 — See Subclause 4.16, “Predefined value types”.

The empty type and the null type are the only immaterial value types.

NOTE 99 — See Subclause 4.16.8, “Immaterial value types: null type and empty type”.

A GQL-implementation may provide additional data types not defined in this document.

NOTE 100 — Examples of such data types are user-defined data types specified by a standard or an application, as well as data types defined by the GQL-implementation.

4.12.3 Data type descriptors

Each data type has an associated data type descriptor. The content of a data type descriptor is 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 element of that data type. In particular, the descriptor of every data type defined in this document includes the declared name of the primary base type of the data type.

A *constraining GQL-object type* is an optional data type characteristic of a descriptor *DESCR* which either constrains instances of the GQL-object type described by *DESCR* or the referents of reference values of the reference value type described by *DESCR*. Instances of such a data type are classified as either open or closed depending the absence or presence, respectively, of the characteristic.

Subclause 18.9, “<value type>”, describes the semantic properties of each value type.

4.12.4 Naming of data types and base types

All data types and base types defined in this document are generally referred to by their *specification names*, which are introduced in Subclause 4.13.1, “Introduction to GQL-object types and related base types”, Subclause 4.14.1, “Introduction to dynamic union types and the dynamic base type”, Subclause 4.15.1, “Introduction to constructed value types and related base types”, and Subclause 4.16.1, “Introduction to predefined value types and related base types”.

Every data type defined in this document is specified by any of its *declared names* (including its *preferred name*, which can be contingent on implementation-defined provisions) as well as by all relevant additional

metadata (such as, e.g., the maximum length of values of certain value types). The declared name of a data type is a non-empty sequence of keywords specified in this document.

Every base type defined in this document is specified by its declared name. The declared name of a base type is a non-empty sequence of keywords that specifies the base type in specific contexts (such as the declared names of base types included in certain data type descriptors) and comprises a declared base type name prefix that is followed by either the word DATA or the word REFERENCE.

4.12.5 Material, nullable, and immaterial data types

Subclause 3.9, “Type terms and definitions”, defines a material data type as a data type excluding the null value. Every GQL-object type defined in this document is a material data type. Two material static data types of different primary base types are always disjoint.

Subclause 3.9, “Type terms and definitions”, defines a nullable data type as including the null value. Every nullable data type defined in this document is a value type. The nullable variant and the material variant of a data type *DT*, are the data types comprising the same material values as *DT* together with or without, respectively, the null value, if existing.

NOTE 101 — See Subclause 4.4.5, “Material values and the null value”.

Subclause 3.9, “Type terms and definitions”, defines an immaterial data type as excluding material values.

NOTE 102 — See Subclause 4.16.8, “Immaterial value types: null type and empty type”.

4.12.6 Most specific static value type and static base type

The *most specific static value type* of a material value *X* is defined as follows. Let *CANDTSET* be the set of all value types of the static base type *BT* of *X* that are supported by the GQL-implementation and that include *X*. The Syntax Rules of Subclause 22.20, “Determination of value type precedence”, are applied with *CANDTSET* as *NDTSET*; let *NDTLIST* be the *NDTLIST* returned from the application of those Syntax Rules. The most specific static value type of *X* is well-defined to be the first element of the list *NDTLIST*. The most specific static value type of the null value is defined as follows. If the GQL-implementation supports Feature GV71, “Immaterial value types: null type support”, then the most specific static value type of the null value is the null type; otherwise, the most specific static value type of the null value is an implementation-defined (ID085) static value type.

The definition of the most specific static value type is based on the notion of the static base type of a value. The static base type of a material value *X* is the common primary base type of all static value types that include *X*. The static base type of all material values defined in this document is well-defined and shall remain unchanged for existing and always be well-defined for any additional material values introduced by a GQL-implementation extended with additional values and value types. The static base type of the null value is the primary base type of the most specific static value type of the null value.

4.12.7 Open and closed data types

The data types of certain primary base types are classified as either *closed* or *open* data types depending on whether they are specified with or without, respectively, explicitly providing certain metadata (such as, e.g., the field types of a record type). This classification is independent of the nullability of data types, i.e., a nullable data type is open or closed if and only if its material variant is open or closed, respectively, too. If a data type *DT* is open, then either *DT* or the nullable variant of *DT* (if existing) is a super type of all data types of the primary base type of *DT*.

This document explicitly classifies the following value types as (nullable) open value types, together with their material, nullable closed, and material closed variants:

- The open dynamic union type, which is specified without explicitly providing metadata indicating its component types.

- The open graph, node, and edge reference value types, which are specified without explicitly providing metadata indicating their constraining GQL-object types.
- The open list value type, which is specified without explicitly providing metadata indicating its maximum cardinality or its element type.

NOTE 103 — The list element type of an open list value type is the open dynamic union type.

- The open record type, which is specified without explicitly providing metadata indicating its field types.

4.12.8 Additional terminology related to data types

An atomic data type is a data type comprising only values that are not composed of values of other data types. The existence of an operation (such as TRIM) that is capable of selecting part of an element of some data type T (such as a character string or a datetime value) does not imply that T is not an atomic data type.

Two data types, $T1$ and $T2$, are said to be compatible if $T1$ is assignable to $T2$, $T2$ is assignable to $T1$, and their descriptors include the same data type name and the same indication regarding the inclusion of the null value, if any. Additionally,

Case:

- 1) If they are record types, it shall further be the case that the value types of their corresponding field types are pairwise compatible or that both are open record types.
- 2) If they are list value types, it shall further be the case that their list element types are compatible.
- 3) If they are reference value types, it shall further be the case that both have a constraining GQL-object type and their constraining GQL-object types are compatible or that both have no constraining object type.
- 4) If they are dynamic union types, then there shall exist a mapping from each component type $FCT1$ of $T1$ to a component type $FCT2$ of $T2$ such that $FCT1$ is assignable to $FCT2$ and a reverse mapping from each component type $RCT2$ of $T2$ to a component type $RCT1$ of $T1$ such that $RCT2$ is assignable to $RCT1$.

4.13 GQL-object types

4.13.1 Introduction to GQL-object types and related base types

The GQL language defines the following GQL-object object types: *graph types*, *node types* (alternatively: *vertex types*), *edges types* (alternatively: *relationship types*), and *binding table types*.

NOTE 104 — Property types and field types are not considered GQL-object types.

The declared names of static base types of GQL-object types defined in this document are:

- GRAPH DATA, which specifies the *graph type base type*. The graph type base type is the base type that comprises all graph types.
- The implementation-defined (ID090) choice of NODE DATA or VERTEX DATA, which specifies the *node type base type*. The node type base type is the base type that comprises all node types.
- The implementation-defined (ID091) choice of EDGE DATA or RELATIONSHIP DATA, which specifies the *edge type base type*. The edge type base type is the base type that comprises all edge types.
- BINDING TABLE DATA, which specifies the *binding table type base type*. The binding table type base type is the base type that comprises all binding table types.

For reference purposes:

- The GQL-object types specified with the leading keyword GRAPH are referred to as graph types and the elements of graph types are referred to as graphs or property graphs. The primary base type of all graph types is the graph type base type.
- The GQL-object types specified with the leading keywords NODE or VERTEX are referred to as node types or, alternatively, as vertex types and the elements of node types are referred to as nodes or, alternatively, as vertices. The primary base type of all node types is the node type base type.
- The GQL-object types specified with the leading keywords EDGE or RELATIONSHIP are referred to as edge types or, alternatively, as relationship types and the elements of edge types are referred to as edges or, alternatively, as relationships. The primary base type of all edge types is the edge type base type.
- The GQL-object types specified with the leading keywords BINDING TABLE or TABLE are referred to as binding table types and the elements of binding table types are referred to as binding tables. The primary base type of all binding table types is the binding table type base type.

4.13.2 Graph types and graph element types

4.13.2.1 Introduction to graph types and graph element types

Subclause 3.9, “Type terms and definitions” defines a graph type as a GQL-object type describing a graph in terms of restrictions on its labels, properties, nodes, edges, and topology. The purpose of a graph type is to constrain the set of nodes and edges that can be contained in a graph.

A graph type is created in a GQL-schema and is identified by a <graph type name>. A graph type is a primary object.

A graph is of a graph type (i.e., is included in that graph type) if each of the nodes in the graph are of a node type specified in the graph type and each of the edges in the graph are of an edge type specified in the graph type. Inserting or updating nodes and edges in a graph that has a constraining graph type such that the graph would no longer be of that graph type causes an exception condition to be raised: *graph type violation (G2000)*.

Multiple graphs may reference the same graph type.

A graph type comprises:

- A node type set, which is a set comprising zero or more node types.
- An edge type set, which is a set comprising zero or more edge types.

4.13.2.2 Graph type descriptors

A graph type is described by its graph type descriptor that comprises:

- The declared name of the primary base type of all graph types (GRAPH DATA).
- The preferred name of graph types (implementation-defined (ID089) choice of GRAPH or PROPERTY GRAPH).
- The set of node type descriptors (also known as a node type set).
- The set of edge type descriptors (also known as an edge type set).
- The node type key label set dictionary that maps node type key label sets to node types contained in the node type set of this graph type descriptor.
- The edge type key label set dictionary that maps edge type key label sets to sets of edge types contained in the edge type set of this graph type descriptor.

NOTE 105 — Without Feature GG25, “Relaxed key label set uniqueness for edge types”, the edge type key label set dictionary maps every edge type key label set to a singleton set of edge types.

Two graph type descriptors describe equal graph types if they contain:

- Equal node type sets.
- Equal edge type sets.
- Equal node type key label set dictionaries.
- Equal edge type key label set dictionaries.

Two node and edge type key label set dictionaries are equal if and only if they:

- Map equal sets of node type key label sets and edge type key label sets,
- Map each of these node type key label sets to equal node types, and
- Map each of these edge type key label sets to equal sets of edge types.

For every graph type *GT* that is a catalog object, the graph type descriptor of *GT* is a catalog object descriptor. The name of *GT* (also known as graph type name) is the catalog object name of *GT*. *GT* is a *named graph type*.

4.13.2.3 Node types

A *node type* is the data type of nodes that have specific labels and that have specific properties and whose properties have a specific property value type.

Each node type comprises:

- A node type label set that comprises a set of zero or more labels. A label has a name that is an identifier that is unique within the node type.
- A node type property type set that comprises a set of zero or more property types.

NOTE 106 — See Subclause 4.13.2.5, “Property types”.

NOTE 107 — The names of node type labels and of node type property types are in separate namespaces. That is, a label and a property type can have the same name in a node type.

A GQL-implementation is permitted to regard two <node type specification>s as equivalent, if they have the same node type label set and node type property type set, as permitted by the Syntax Rules of Subclause 18.4, “<label set phrase> and <label set specification>”, and Subclause 18.5, “<property types specification>”. When two or more <node type specification>s are equivalent, the GQL-implementation chooses one of these equivalent <node type specification>s as the normal form representing that equivalence class of <node type specification>s. The normal form determines the preferred name of the node type in data type descriptors.

A node *N* in a graph is of a node type *NT* (i.e., is included in *NT*) if the label set of *N* and the node type label set of *NT* are the same, the number of properties of *N* and the number of property types of *NT* are the same, and every property of *N* is of a property type of *NT*.

A node type is described by the node type descriptor. The node type descriptor comprises:

- The declared name of the primary base type of all node types (NODE DATA).
- The preferred name of node types (implementation-defined (ID090) choice of NODE or VERTEX).
- The set of zero or more labels (also known as a node type label set). A label has a name that is an identifier that is unique within the node type label set.

The minimum cardinality of node type label sets is the implementation-defined (IL001) minimum cardinality of node label sets.

The maximum cardinality of node type label sets is the implementation-defined (IL001) maximum cardinality of node label sets.

- The set of zero or more property type descriptors (also known as a node type property type set).

The maximum cardinality of node type property type sets is the implementation-defined (IL002) maximum cardinality of node property type sets.

Two node type descriptors describe equal node types if they contain equal node type label sets and equal node type property type sets.

4.13.2.4 Edge types

An *edge type* is the data type of edges that have specific labels and that have specific properties and whose properties have a specific property value type and whose endpoints conform to specific node types.

Each edge type comprises:

- An edge type label set that comprises a set of zero or more labels. A label has a name that is an identifier that is unique within the edge type.
- An edge type property type set that comprises a set of zero or more property types.
- NOTE 108 — See Subclause 4.13.2.5, “Property types”.
- Two (possibly identical) endpoint node types that are node types contained in the same graph type.
- The indication of whether the edge type is a directed edge type or an undirected edge type (which is also called the *directionality* of the edge type).

Additionally, a directed edge type identifies one of its endpoint node types as its source node type, and the other as its destination node type. The direction of a directed edge type is from its source node type to its destination node type.

NOTE 109 — The names of edge type labels and of edge type property types are in separate namespaces. That is, a label and a property type can have the same name in an edge type.

A GQL-implementation is permitted to regard two <edge type specification>s as equivalent, if they are either both directed or undirected, have the same endpoint node types, edge type label sets and edge type property type sets, as permitted by the Syntax Rules of Subclause 18.3, “<edge type specification>”, Subclause 18.4, “<label set phrase> and <label set specification>”, and Subclause 18.5, “<property types specification>”. When two or more <edge type specification>s are equivalent, the GQL-implementation chooses one of these equivalent <edge type specification>s as the normal form representing that equivalence class of <edge type specification>s. The normal form determines the preferred name of the edge type in data type descriptors.

An edge *E* in a graph is of an edge type *ET* (i.e., is included in *ET*) if the label set of *E* and the edge type label set of *ET* are the same, the number of properties of *E* and the number of property types of *ET* are the same, and every property of *E* is of a property type of *ET*, and one of the following is true:

- *E* and *ET* are directed and the source and destination endpoints of *E* are of the source and destination endpoint node types of *ET*, respectively.
- *E* and *ET* are undirected and the binary relation over *E* and *ET* that relates each endpoint in *E* to each of its endpoint node types in *ET* is left-total and right-total.

An edge type is described by the edge type descriptor. The edge type descriptor comprises:

- The declared name of the primary base type of all edge types (EDGE DATA).

- The preferred name of edge types (implementation-defined (ID091) choice of EDGE or RELATIONSHIP).
- The set of zero or more labels (also known as an edge type label set). A label has a name that is an identifier that is unique within the edge type label set.

The minimum cardinality of edge type label sets is the implementation-defined (IL001) minimum cardinality of edge label sets.

The maximum cardinality of edge type label sets is the implementation-defined (IL001) maximum cardinality of edge label sets.

- The set of zero or more property type descriptors (also known as an edge type property type set).
The maximum cardinality of edge type property type sets is the implementation-defined (IL002) maximum cardinality of edge property sets.
- The indication of whether the edge type is directed or undirected.
- Case:

- If the edge type is directed, then:
 - The source node type descriptor.
 - The destination node type descriptor.
- If the edge type is undirected, then the set of one or two endpoint node type descriptors.

NOTE 110 — If the two endpoint node types of an undirected edge type are the same, then the set contains only one endpoint node type descriptor; otherwise, it contains two endpoint node type descriptors.

Two edge type descriptors describe equal edge types if they contain:

- Equal edge type label sets,
- Equal edge type property type sets,
- Equal indications whether they are directed or undirected, and
- Case:

- If the edge types are directed, then:
 - Equal source node types and
 - Equal destination node types.
- If the edge types are undirected, then equal sets of endpoint node types.

4.13.2.5 Property types

A property type is a pair comprising:

- A name, which is an identifier.
- A value type, which can be any supported property value type.

NOTE 111 — See Subclause 4.3.4, “Properties and supported property value types”.

Two property types are equal if they have equal names and equal value types.

Every property type is described by a property type descriptor that comprises:

- A name, which is an identifier.
- A value type, which can be any supported property value type.

4.13.2.6 Key label sets

A key label set is a set of labels. Key label sets are elements of a graph type. An element type of a graph type may be associated with a key label set. An element type that is associated with a key label set is said to be a *keyed* element type. The key label set that a keyed element type *KET* is associated with is said to be the key label set of *KET*. An element type that is not associated with a key label set is said to be a *structural* element type.

Every keyed node type of a graph type is uniquely identified in the keyed node types of its graph type by its key label set. All keyed node types of a graph type appear in the node type key label set dictionary of the graph type's descriptor. The node type key label set dictionary of a graph type maps each node type key label set to the keyed node type associated with that key label set.

If two keyed edge types *KET* and *OKET* of the same graph type have the same key label set, then it holds that *KET* and *OKET* have:

- Equal edge type label sets,
- Equal edge type property type sets, and
- Equal indications of whether they are directed or undirected.

All keyed edge types of a graph type appear in the edge type key label set dictionary of the graph type's descriptor. The edge type key label set dictionary of a graph type maps each edge type key label set to the set of keyed edge types that are associated with that key label set.

If Feature GG25, "Relaxed key label set uniqueness for edge types" is not supported, then every keyed edge type of a graph type is uniquely identified in the keyed edge types of its graph type by its key label set and the edge type key label set dictionary of that graph type maps each edge type key label set to the singleton set comprising the single keyed edge type associated with that key label set.

4.13.2.7 Structural consistency of element types

Graph types can only contain element types that are *structurally consistent* with one another and also can only contain edge types that are *structurally endpoint-consistent* with one another.

NOTE 112 — These consistency requirements guarantee that the element types of a graph type do not have mutually conflicting definitions, as explained in the remainder of this Subclause.

Graph types adhere to the following two consistency criteria to ensure structural consistency of element types and structural endpoint-consistency of edge types:

- *Key label set implication consistency.*
- *Property value type consistency.*

Both consistency criteria rely on the *graph-type specific combination of property value types* to ensure that certain property types of element types of a graph type that have the same name also either have the same value type or, if Feature GG24, "Relaxed structural consistency" is supported, have value types that can be combined by an application of the Syntax Rules of Subclause 22.18, "General combination of value types" to obtain a supported property value type.

NOTE 113 — See Subclause 22.17, "Graph-type specific combination of property value types".

Key label set implication consistency considers element types that imply one another. If the key label set of a keyed node type *SUPERNT* is a subset of the label set of a node type *SUBNT*, then *SUBNT* implies *SUPERNT*. If the key label set of a keyed edge type *SUPERET* is a subset of the label set of an edge type

SUBET and either *SUBET* is structural or the key label sets of *SUBET* and *SUPERET* are different, then *SUBET* implies *SUPERET*.

NOTE 114 — See [Syntax Rule 6\)](#) of Subclause 18.1, “<nested graph type specification>” for the corresponding definition of implication of element type specifications.

If an element type *SUBELT* implies another element type *SUPERELT*, then they are structurally consistent, i.e.,

- Both are node types or both are edge types.
- The label set of *SUPERELT* is a subset of the label set of *SUBELT*.
- The set of names of property types of *SUPERELT* is a subset of the set of names of property types of *SUBELT*.
- The value type of each property type of *SUPERELT* combines to itself with the value type of the corresponding property type of *SUBELT* with the same name under graph-type specific combination of value types.

NOTE 115 — See [Syntax Rule 7\)](#) of Subclause 18.1, “<nested graph type specification>” for the corresponding definition of structurally consistent <node type specification>s and <edge type specification>s.

If an edge type *SUBEDGT* implies another edge type *SUPEREDGT*, then they are structurally endpoint-consistent, i.e.,

- Both are directed and the endpoint node types of *SUBEDGT* are structurally consistent with the respective endpoint node types of *SUPEREDGT*.
- Both are undirected and the binary relation over the endpoint node types of *SUBEDGT* and *SUPEREDGT* that comprises all pairs of endpoint node types of *SUBEDGT* and *SUPEREDGT* whose components are structurally consistent is both left-total and right-total.

NOTE 116 — See [Syntax Rule 15\)](#) of Subclause 18.1, “<nested graph type specification>” for the corresponding definition of structurally endpoint-consistent <edge type specification>s.

Property value type consistency considers element types of a graph type whose property types share names. Property value type consistency applies to all element types of a graph type regardless of whether such element types are keyed or structural. In a property value type consistent graph type *GT*, for every name *PN* of a property type of some element type of *GT*, the result of the graph-type specific combination of all value types of property types whose name is *PN* of element types of *GT* is a supported property value type.

If Feature GG24, “Relaxed structural consistency” is not supported, then all graph types are property value type consistent; otherwise, Feature GG24, “Relaxed structural consistency” is supported and it is possible that some graph types are not property value type consistent.

4.13.3 Binding table types

A binding table type is a GQL-object type that describes the records that can occur in a binding table that is of that binding table type.

Every binding table type is described by a binding table data type descriptor. A binding table data type descriptor comprises:

- The declared name of the primary base type of all binding table types (BINDING TABLE DATA).
- The closed material record type.

For every binding table type *BTT* with a record type *RT*, a binding table *BT* is of *BTT*, if and only if every record *R* contained in *BT* is of *RT*.

In this document, a column is a field type of the record type of a binding table type, a column name is the name of a column, and a column type is the value type of a column. Hence, a column name identifies the fields of the same name of the records of a binding table.

A GQL-implementation is permitted to regard two <binding table type>s as equivalent, if they have the same record type and indication regarding the inclusion of the null value, as permitted by the Syntax Rules of [Subclause 18.8](#), “<binding table type>”. When two or more <binding table type>s are equivalent, the GQL-implementation chooses one of these equivalent <binding table type>s as the normal form representing that equivalence class of <binding table type>s.

A *unit binding table type* is a binding table type whose record type is a unit record type.

The binding table type of an empty binding table is a unit binding table type unless specified otherwise.

4.14 Dynamic union types

4.14.1 Introduction to dynamic union types and the dynamic base type

The *dynamic base type* is the base type whose declared name is ANY DATA and that comprises all value types (including all dynamic union types).

A *dynamic union type* is a value type that comprises values of multiple data types. Therefore dynamic union types can be used to characterize sites capable of being occupied by material values of different static base types (e.g., can hold both exact numbers and character strings). The primary base of all dynamic union types is the dynamic base type.

The conditional application of a General Rule may depend on determining the static base type or the most specific static value type of a value assigned to a site whose declared type is a dynamic union type, e.g., in order to ensure that certain assignments do not violate the declared types of their target sites.

NOTE 117 — Most <value expression>s defined in this document are oblivious to the existence of dynamic union types; only certain parts of this document related to, e.g., comparison, testing, casting, and store assignment, include additional provisions for correctly handling dynamic union types. See [Subclause 4.14.4](#), “[Dynamic generation of type tests and casts](#)” regarding the use of <value expression>s with or without operands whose declared type is a dynamic union type.

4.14.2 Dynamic union data type descriptors

Every dynamic union type is described by its dynamic union data type descriptor. A dynamic union data type descriptor comprises:

- The declared name of the primary base type of all dynamic union types (ANY DATA).
- The preferred name of the dynamic union type.
- The indication of whether the type is open or closed.
- The component types, a finite set of two or more data types.

NOTE 118 — It is possible that a syntactic transformation generates a <dynamic union type> that has only exactly one component type. However, such a <dynamic union type> is always replaced by its normal form, i.e., by the normal form of that component type. See [Subclause 4.14.3](#), “[Characteristics of dynamic union types](#)”.

- The indication of whether the dynamic union type includes the null value.

4.14.3 Characteristics of dynamic union types

The open nullable dynamic union type is the dynamic union type that comprises all values. More generally, a dynamic union type *DUT* comprises all values included in at least one of the *component types* of *DUT*.

The set of component types of the open dynamic union type is implementation-defined (IV012) such that for every static base type, at least one largest supertype of each of the base type's data types supported by the GQL-implementation is included. The set of component types of a closed dynamic union type is specified explicitly.

The indication of whether a dynamic union type *DUT* includes the null value is defined as follows. If at least one of the component types of *DUT* is nullable, then the indication is *True*; otherwise, it is *False*.

A GQL-implementation regards certain <dynamic union type>s as equivalent, if they have the same component types and indications regarding the inclusion of the null value, as permitted by Subclause 18.9, "<value type>". When two or more <dynamic union type>s are equivalent, the GQL-implementation chooses one of these equivalent <dynamic union type>s as the normal form representing that equivalence class of <dynamic union type>s.

NOTE 119 — The normal form defined in Subclause 18.9, "<value type>", ensures that effectively every component type is a static value type and that every dynamic union type whose component types are the same as the component types of the open dynamic union types are considered equivalent to the open dynamic union type that has the same nullability characteristic.

Furthermore, a <dynamic union type> whose component type list comprises a single static value type *SVT* is regarded as equivalent to the normal form of *SVT*.

A value *V* is assignable to a site whose declared type is a dynamic union type *DUT* if and only if *V* is assignable to a component type of *DUT*. If store assignment of a value to a site of a dynamic union type is not possible without loss of information, then an exception condition may be raised.

NOTE 120 — Every value can be assigned to a site whose declared type is the open dynamic union type.

4.14.4 Dynamic generation of type tests and casts

4.14.4.1 Introduction to dynamic generation of type tests and casts for <value expression>s

This document primarily defines the GQL language using a static typing discipline. This may be extended through the use of dynamic union types, as defined in Subclause 4.14, "Dynamic union types". However, most Syntax Rules stated in this document that enforce restrictions on the declared type of <value expression>s do not particularly consider <value expression>s whose declared type is either a dynamic value type or an open value type.

In order to support GQL-implementations that prefer the use of a dynamic typing discipline, if a hypothetical application of all relevant Syntax Rules that are not Conformance Rules to a given <GQL-program> *PROGRAM* of a GQL-request would fail with a syntax error caused by an unmet requirement on the declared type of an expression being of a dynamic type or an open value type, then instead the GQL-implementation may execute a cured variant of *PROGRAM* obtained by the following syntactic transformations.

- Dynamic generation of type tests and strict casts for a <value expression> without operands.
- Dynamic generation of type tests and strict casts for a <value expression> with operands.
- Dynamic generation of additional type tests and lax casts for a <value expression>.

4.14.4.2 Dynamic generation of type tests and strict casts for a <value expression> without operands

Let *PROGRAM* be the <GQL-program> and let *VE* be a <value expression> contained in *PROGRAM* that has no operands and whose declared type *DOT* is either a dynamic value type or an open value type.

If a hypothetical application of all relevant Syntax Rules to *PROGRAM* would fail due to *DOT* not meeting the requirements of the syntactic context of *VE*, then *PROGRAM* may be cured by replacing *VE* as follows.

- 1) Let *CANDTYPES* be the set of static value types that meet the requirements of the syntactic context of *VE* on *DOT*.
- 2) The Syntax Rules of Subclause 22.20, “Determination of value type precedence”, are applied with *CANDTYPES* as *NDTSET*; let *VETYPES* be the *NDTLIST* returned from the application of those Syntax Rules.
- 3) Let *ARMTYPES* be the subsequence of *VETYPES* that contains every type *VT* from *VETYPES* such that:
 - a) *VT* is a subtype of *DOT*.
 - b) The following shall be valid according to the Syntax Rules of Subclause 20.8, “<cast specification>”:

CAST (*VE* AS *VT*)

- 4) Let *N* be the number of elements of *ARMTYPES*.
- 5) Let *VAR* be a new system-generated regular identifier.
NOTE 121 — See Syntax Rule 23) of Subclause 21.3, “<token>, <separator>, and <identifier>” for detailed provisions regarding the construction of system-generated regular identifiers.

- 6) For each *i*-th *ARMTYPE_i* in *ARMTYPES*, $1 \text{ (one)} \leq i \leq N$, let *VE_i* be:

CAST (*VAR* AS *ARMTYPE_i*)

- 7) Let *ERROR* be an implementation-dependent (UV003) <value expression> whose evaluation raises the exception condition: *data exception — invalid value type (22G12)*.
- 8) If *N* is zero, *VE* is replaced by *ERROR*; otherwise, it is replaced by:

```
LET VAR=VE IN
CASE VAR
  WHEN IS TYPED ARMTYPE1 THEN VE1
  WHEN IS TYPED ARMTYPE2 THEN VE2
  ...
  WHEN IS TYPED ARMTYPEN THEN VEN
  ELSE ERROR
END
END
```

NOTE 122 — Subclause 4.14.4.4, “Dynamic generation of additional type tests and lax casts for a <value expression>”, specifies additional provisions that allow a GQL-implementation to further amend the <case expression> generated above.

4.14.4.3 Dynamic generation of type tests and strict casts for a <value expression> with operands

Let *PROGRAM* be the <GQL-program> and let *VE* be a <value expression> contained in *PROGRAM* that has *NOPS* operands *OP₁*, ..., *OP_{NOPS}* whose respective declared types are *DOPT₁*, ..., *DOPT_{NOPS}*.

If a hypothetical application of all relevant Syntax Rules to *PROGRAM* would fail due to at least one of the declared types of operands of *VE* that is a dynamic value type or an open value type being ill-typed, i.e., not meeting the requirements of the syntactic context of its respective operand, then *PROGRAM* may be cured by replacing *VE* as follows.

- 1) For *i*, $1 \text{ (one)} \leq i \leq NOPS$, let *OPTYPES_i* be determined as follows.

Case:

ISO/IEC 39075:2024(en)
4.14 Dynamic union types

- a) If OP_i is one of the ill-typed operands, then:
 - i) Let $CANDTYPES_i$ be the set of value types that meet the requirements of the syntactic context of OP_i .
 - ii) The Syntax Rules of [Subclause 22.20, “Determination of value type precedence”](#), are applied with $CANDTYPES_i$ as $NDTSET$; let $OPTYPES_i$ be the $NDTLIST$ returned from the application of those Syntax Rules.
- b) Otherwise, let $OPTYPES_i$ be a list containing the null value.

- 2) For i , $1 \text{ (one)} \leq i \leq NOPS$, let VAR_i be a new system-generated regular identifier that is not contained in *PROGRAM* and not used for naming a catalog object in the GQL-catalog and for all j , $1 \text{ (one)} \leq i < j \leq NOPS$, VAR_i and VAR_j are not equivalent.

NOTE 123 — See [Syntax Rule 23](#)) of [Subclause 21.3, “<token>, <separator>, and <identifier>”](#) for detailed provisions regarding the construction of system-generated regular identifiers.

- 3) Let *COMBOS* be the ordered cross product $OPTYPES_1 \times \dots \times OPTYPES_{NOPS}$ i.e., a list of lists of length *NOPS* such that *COMBOS* is lexicographically ordered left-to-right as determined by its operands understood as defining the ordered alphabet of symbols at each respective position. Let *NCOMBOS* be the cardinality of *COMBOS*.
- 4) Let *ARMLIST* be the list of <simple case>s that are determined as follows.

Case:

- a) Initially, *ARMLIST* is the zero-length character string.
- b) For j , $1 \text{ (one)} \leq j \leq NCOMBOS$, and for i , $1 \text{ (one)} \leq i \leq NOPS$, let $COMBOTYPE_{j,i}$ be the i -th list element of the j -th combination of *COMBOS*.
- c) For j , $1 \text{ (one)} \leq j \leq NCOMBOS$:
 - i) For i , $1 \text{ (one)} \leq i \leq NOPS$, let $COND_i$ and COP_i be defined as follows:
 - 1) Let $COPTYPE$ be $COMBOTYPE_{j,i}$.
 - 2) Let $COND_i$ be defined as follows. If $COPTYPE$ is the null value, then $COND_i$ is TRUE; otherwise, $COND_i$ is

$$(\text{VAR}_i \text{ IS } COPTYPE)$$
 - 3) Let COP_i be defined as follows. If $COPTYPE$ is the null value, then COP_i is VAR_i ; otherwise, VAR_i is

$$\text{CAST } (\text{VAR}_i \text{ AS } COPTYPE)$$
 - ii) For i , $1 \text{ (one)} \leq i \leq NOPS$, let $COND$ be the AND-concatenation of all $COND_i$:

$$COND_1 \text{ AND } COND_2 \dots \text{ AND } COND_{NOPS}$$
 - iii) For i , $1 \text{ (one)} \leq i \leq NOPS$, let NVE be VE with every OP_i replaced by COP_i .
 - iv) Let *ARM* be:

$$\text{WHEN } COND \text{ THEN } NVE$$

- v) If a hypothetical application of the Syntax Rules of *NVE* in the same syntactic context as *VE* would succeed, then *ARM* is appended to *ARMLIST*.

NOTE 124 — This verifies the validity of the generated <value expression>.

- 5) Let *ERROR* be an implementation-dependent (UV003) <value expression> whose evaluation raises the exception condition: *data exception — invalid value type* (22G12).
- 6) If *ARMLIST* is the zero-length character string, then *VE* is replaced by *ERROR*; otherwise, it is replaced by:

```
LET VAR1=OP1, . . . , VARNOPS=OPNOPS IN  
CASE  
  ARMLIST  
ELSE ERROR  
END  
END
```

NOTE 125 — Subclause 4.14.4.4, “Dynamic generation of additional type tests and lax casts for a <value expression>”, specifies additional provisions that allow a GQL-implementation to further amend the <case expression> generated above.

4.14.4.4 Dynamic generation of additional type tests and lax casts for a <value expression>

The provisions regarding the dynamic generation of type tests and casts for a <value expression> with or without operand only generate strict casts, i.e., casts to one of the value types of its argument value.

A GQL-implementation may also generate lax casts, i.e., casts from a <value expression> *VE* contained in *PROGRAM* to a value type *VT* that is not including the result of *VE* as long as *CAST(VE AS VT)* is valid according to the Syntax Rules of Subclause 20.8, “<cast specification>”. The exact manner in which lax casts (and supporting type tests) are generated and included in the syntax transforms for the dynamic generation of strict casts is implementation-defined (IW018).

4.15 Constructed value types

4.15.1 Introduction to constructed value types and related base types

The GQL language defines the following kinds of constructed value types: path value types, list value types, and record types.

The declared names of static base types of constructed value types defined in this document are:

- *PATH VALUE DATA*, which specifies the *path value type base type*. The path value type base type is the base type that comprises all path value types.
- *LIST VALUE DATA*, which specifies the *list value type base type*. The list value type base type is the base type that comprises all list value types.
- *RECORD DATA*, which specifies the *record type base type*. The record type base type is the base type that comprises all record types.

For reference purposes:

- The value types specified with the leading keyword *PATH* are referred to as *path value types* and the values of path value types are referred to as *path values*. The primary base type of all path value types is the path value type base type.
- The value types specified with the leading keywords *LIST* or *ARRAY* are referred to as *list value types* and the values of list value types are referred to as *list values*. The value types specified with the leading keywords *GROUP LIST* or *GROUP ARRAY* are referred to as *group list value types* and

their values are referred to as *group list values*. All other list value types are referred to as *regular list value types* and their values are referred to as *regular list values*. The primary base type of all list value types is the list value type base type.

- The value types specified with the leading keyword RECORD are referred to as *record types* and the values of record types are referred to as *records*. The primary base type of all record types is the record type base type.

4.15.2 Path value types

A material value of a path value type is a path value. A path value *PV* encapsulates a graph element reference value list value that is called the *path element list* of *PV*. In this document, a *single-node path value* is a path value whose path element list comprises a single node reference value.

NOTE 126 — A single-node path value is frequently called an “empty path”; the latter term has been avoided because of possible confusion of its underlying path element list with an empty list value, which has no elements (neither node reference values nor edge reference values).

If the sequence comprising the nodes and edges referenced by the elements of a graph element reference value list value in the order in which they appear in the sequence is a path (in the sense of Subclause 4.11.2, “Paths”), then the sequence is said to *identify a path*. For every path value *PV* constructed in this document it holds that if the path element list *PEL* of *PV* excludes invalidated graph element reference values, then *PEL* always identifies a path.

Every path value type *PVT* is described by a path value data type descriptor. A path value data type descriptor comprises:

- The declared name of the primary base type of all path value types (PATH VALUE DATA).
- The indication of whether the type contains the null value.

The cardinality of a path value *PV* of *PVT* is the cardinality of its path element list *PEL*, which is always less than or equal to the implementation-defined (IL015) maximum cardinality of path element lists of path value types. Furthermore, the length of *PV* is defined as follows: If *PEL* contains at least one node reference value, then the length of *PV* is the number of elements of *PEL* that are edge reference values; otherwise, the length of *PV* is the null value.

A path value *P* is assignable to a site of at least the material path value type.

Any two path values are essentially comparable values.

4.15.3 List value types

A material value of a *list value type* is a *list value*. A list value *L* is an ordered collection of elements, in which each element is associated with exactly one ordinal position in *L*. Further, the *position offset* of an element of *L* at ordinal position *p* is *p*−1. If *N* is the cardinality of *L*, then the ordinal position *p* of an element is a positive integer in the range $1 \text{ (one)} \leq p \leq N$ and the position offset *o* of an element is a non-negative integer in the range $0 \text{ (zero)} \leq o < N$.

Every list value type is described by a list value data type descriptor. A list value data type descriptor comprises:

- The declared name of the primary base type of all list value types (LIST VALUE DATA).
- The preferred name of the specific list value type, which is the declared name of the normal form of the list value type.
- The *list element type* of all possible values that list values of the list value type may contain.
- The indication of whether the type is a *group list value type* or a *regular list value type*. A group list value type is a list value type whose values are known to directly or indirectly originate from an

element variable that has group degree of reference, as detailed in [Subclause 4.11.5, “References to graph pattern variables”](#). A regular list value type is a list type that is not a group list value type.

- The maximum cardinality of the list value type.
- The indication of whether the type contains the null value.

If a list value type is a group list value type, then its list element type is either a node reference value type, an edge reference value type, or a closed dynamic union type whose component types comprise either only node reference value types or only edge reference value types.

For a list value type *LVT*, the indication of whether *LVT* is a group list value type or a regular list value type is also referred to as the *group characteristic* of *LVT*. The *regular variant* of a group list value type *GLVT* is a regular list value type that has the same characteristics as *GLVT* except for the group characteristic.

The maximum cardinality of a list value type is a positive integer and the maximum cardinality of a list value type with list element type *LET* is less than or equal to the implementation-defined (IL015) maximum cardinality for list value types whose list element type is *LET*.

For every list value type *LT*, a list value *L* is a material value of *LT*, if and only if the cardinality of *L* does not exceed the maximum cardinality of *LT* and every element of *L* is of the list element type of *LT*.

A GQL-implementation is permitted to regard certain <list value type>s as equivalent, if both have the same list element type and the same indication regarding the inclusion of the null value, as permitted by the Syntax Rules of [Subclause 18.9, “<value type>”](#). When two or more <list value type>s are equivalent, the GQL-implementation chooses one of these equivalent <list value type>s as the normal form representing that equivalence class of <list value type>s.

A list value *L* whose declared type *DT* is assignable to every site that is at least of some material list value type *LVT* for which it holds that the cardinality of *L* does not exceed the maximum cardinality of *LVT*, the list element type of *DT* is assignable to the list element type of *LVT*, and if *DT* is a group list value type, then *LVT* is a group list value type.

Two list values are essentially comparable values if and only if their elements are essentially comparable values.

In this document, an *empty list value* is a list value of cardinality zero.

4.15.4 Record types

A material value of a record type is a record. A record is a set of fields, each such field has a name, which is a character string, and a value. The names of fields of records that are specified by <identifier>s are the canonical name forms of those <identifier>s. The record with zero fields is called the *unit record*.

Every record type is described by a record data type descriptor. A record data type descriptor comprises:

- The declared name of the primary base type of all record types (RECORD DATA).
- The indication of whether the type is closed or open.
- If the record type is closed, then a (possibly empty) set of field types. Each field type is a pair comprising a name, which is an identifier, and a value type. The name of each field type is unique within the record type.
- The indication of whether the type contains the null value.

The cardinality of a record is the cardinality of the set of fields of the record. The cardinality of a closed record type is the cardinality of the set of field types of the record type. The cardinality of a record type shall be less than or equal to the implementation-defined (IL015) maximum number of record fields.

ISO/IEC 39075:2024(en)
4.15 Constructed value types

Every field type of a record type is described by a field type descriptor that comprises:

- A name, which is an identifier.
- A value type.

The names of field types of record types that are specified by <identifier>s are the canonical name forms of those <identifier>s.

For every record type *RT*, a record *R* is a material value of *RT*, if and only if either *RT* is open or all the following conditions are true:

- *RT* is closed.
- *R* has the same number of fields as *RT* has field types.
- For every field type in *RT* with name *FN* and value type *FVT*, *R* has a field whose name is *FN* and whose value is of *FVT*.

A *unit record type* is a record type that has zero field types. The only material value of a unit record type is the unit record.

A GQL-implementation is permitted to regard two <record type>s as equivalent, if they have the same indication regarding the inclusion of records with additional fields, set of field types, if present, and indication regarding the inclusion of the null value, as permitted by the Syntax Rules of [Subclause 18.9](#), "<value type>". When two or more <record type>s are equivalent, the GQL-implementation chooses one of these equivalent <record type>s as the normal form representing that equivalence class of <record type>s.

Two records or record types *A* and *B* are called *field name-equal* if and only if there is a bijection that maps every name of *A* to an equal name of *B* and every name of *B* to an equal name of *A*. Similarly, *A* and *B* are called *field name-disjoint* if and only if there are no field names *F_A* of *A* and *F_B* of *B* such that *F_A* and *F_B* are equal.

Two closed record types *RT1* and *RT2* are *field type-combinable* if and only if all of the following hold:

- *RT1* and *RT2* are comparable value types.
- For every pair of field types *FT1* from *RT1* and *FT2* from *RT2* with equal names, the Syntax Rules of [Subclause 22.18](#), "General combination of value types", can be applied successfully with the set comprising the value types of *FT1* and *FT2* as *DTSET*; returning *DT* as the *RESTYPE*.

The *combined field types* *CFT* of two field type-combinable record types *RT1* and *RT2* are determined as follows:

- 1) Let *FTU* be the union of the field types of *RT1* and *RT2*.
- 2) For every non-empty subset of all field types *FTP* of *FTU* whose names are equal:
 - a) Let *NAME* be the name of one of the field types in *FTP*.
 - b) The Syntax Rules of [Subclause 22.18](#), "General combination of value types", are applied with *FTP* as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those Syntax Rules.
 - c) The field type with name *NAME* and value type *DT* is added to *CFT*.

Given two records *R1* and *R2*, the phrase *R1 amended with R2* or similar grammatical variants denotes a new record that comprises:

- Every field of *R1* whose name differs from the name of every field of *R2*.
- Every field of *R2*.

ISO/IEC 39075:2024(en)
4.15 Constructed value types

NOTE 127 — The definition gives precedence to a field $F2$ in $R2$ over a field $F1$ in $R1$ if both fields have the same name.

Similarly, given two closed record types $RT1$ and $RT2$, the phrase *RT1 amended with RT2* or similar grammatical variants denotes a new record type RT that comprises:

- Every field type of $RT1$ whose name differs from the name of every field type of $RT2$.
- Every field type of $RT2$.

NOTE 128 — The definition gives precedence to a field type $FT2$ in $RT2$ over a field type $FT1$ in $RT1$ if both field types have the same name.

NOTE 129 — If $R1$ and $R2$ are records whose closed record types are $RT1$ and $RT2$, respectively, then $R1$ amended with $R2$ is included in $RT1$ amended with $RT2$.

If $RT1$ and $RT2$ are material, then RT is material; otherwise, RT is nullable.

Let R be a record, let RT be a closed record type, and let FS be a set of (field) names. The phrase *R restricted to the fields identified by FS* or similar grammatical variants denotes a new record that comprises every field of R whose name is in FS . By extension, the phrase *RT restricted to the fields identified by FS* or similar grammatical variants denotes a new record type that comprises every field type of RT whose name is in FS . If R is included in RT , then R is restricted to the fields identified by FS is included in RT restricted to the fields identified by FS .

Similarly, the phrase *R without the fields identified by FS* or similar grammatical variants denotes a new record that comprises every field of R whose name is not in FS . By extension, the phrase *RT without the fields identified by FS* or similar grammatical variants denotes a new record type that comprises every field type of RT whose name is not in FS . If R is included in RT , then R without the fields identified by FS is included in RT without the fields identified by FS .

Two records are essentially comparable values if and only if the two records are field-name equal and their respective field values are essentially comparable values.

A record R is assignable to a site that is at least of a some material record type RT for which it holds that either RT is open or it holds that the set of field names of R and the set of field type names of RT are the same and every field of R is assignable to the value type of the field type with the same name of RT .

4.16 Predefined value types

4.16.1 Introduction to predefined value types and related base types

The GQL language defines the following kinds of predefined value types: *Boolean types*, *character string types*, *byte string types*, *numeric types*, *temporal types*, and reference value types for references to binding tables, graphs, nodes, or edges.

The declared names of static base types of predefined value types defined in this document are:

- **BOOLEAN DATA**, which specifies the *Boolean type base type*. The Boolean type base type is the base type that comprises all Boolean types.
- **STRING DATA**, which specifies the *character string type base type*. The character string type base type is the base type that comprises all character string types.
- **BINARY DATA**, which specifies the *byte string type base type*. The byte string type base type is the base type that comprises all byte string types.
- **EXACT NUMERIC DATA**, which specifies the *exact numeric type base type*. The exact numeric type base type is the base type that comprises all exact numeric types.
- **FLOAT NUMERIC DATA**, which specifies the *approximate numeric type base type*. The approximate numeric type base type is the base type that comprises all approximate numeric types.

ISO/IEC 39075:2024(en)
4.16 Predefined value types

- TEMPORAL INSTANT DATA, which specifies the *temporal instant type base type*. The temporal instant type base type is the base type that comprises all temporal instant types.
- TEMPORAL DURATION DATA, which specifies the *temporal duration type base type*. The temporal duration type base type is the base type that comprises all temporal duration types.
- GRAPH REFERENCE, which specifies the *graph reference value type base type*. The graph reference value type is the base type that comprises all graph reference value types.
- TABLE REFERENCE, which specifies the *binding table reference value type base type*. The binding table reference value type is the base type that comprises all binding table reference value types.
- The implementation-defined (ID090) NODE REFERENCE or VERTEX REFERENCE, which specifies the *node reference value type base type*. The node reference value type is the base type that comprises all node reference value types.
- The implementation-defined (ID091) EDGE REFERENCE or RELATIONSHIP REFERENCE, which specifies the *edge reference value type base type*. The edge reference value type is the base type that comprises all edge reference value types.
- NULL DATA, which specifies the immaterial value type base type. The immaterial value type is the base type that comprises all immaterial value types.

For reference purposes:

- The value types specified with the leading keyword BOOL and BOOLEAN are referred to as *Boolean types* and the values of Boolean types are referred to as *truth values*, or, alternatively, as *Booleans*. The primary base type of all Boolean types is the Boolean type base type.
- The value types specified with the leading keyword STRING, CHAR, and VARCHAR are referred to as *character string types* and the values of character string types are referred to as *character strings*. The primary base type of all character string types is the character string type base type.
- The value types specified with the leading keyword BYTES, BINARY, and VARBINARY are referred to as *byte string types* and the values of byte string types are referred to as *byte strings*. The primary base type of all byte string types is the byte string type base type.
- Exact numeric types and approximate numeric types are collectively referred to as *numeric types*. Values of numeric types are referred to as *numbers*. The signed exact numeric types and the unsigned exact numeric types are collectively referred to as *exact numeric types*. Values of exact numeric types are referred to as *exact numbers*. The primary base type of all exact numeric types is the exact numeric type base type.
- The value types specified with the leading keywords DECIMAL, DEC, SMALLINT, SMALL INTEGER, SIGNED SMALL INTEGER, INT, INTEGER, SIGNED INTEGER, INT16, INTEGER16, SIGNED INTEGER16, INT32, INTEGER32, SIGNED INTEGER32, INT64, INTEGER64, SIGNED INTEGER64, INT128, INTEGER128, SIGNED INTEGER128, INT256, INTEGER256, SIGNED INTEGER256, BIGINT, BIG INTEGER, and SIGNED BIG INTEGER are collectively referred to as *signed exact numeric types*. Values of signed exact numeric types are referred to as *signed exact numbers*.
- The value types specified with the leading keywords USMALLINT, UNSIGNED SMALL INTEGER, UINT, UNSIGNED INTEGER, UINT16, UNSIGNED INTEGER16, UINT32, UNSIGNED INTEGER32, UINT64, UNSIGNED INTEGER64, UINT128, UNSIGNED INTEGER128, UINT256, UNSIGNED INTEGER256, UBIGINT, and UNSIGNED BIG INTEGER are collectively referred to as *unsigned exact numeric types*. Values of unsigned exact numeric types are referred to as *unsigned exact numbers*.
- Exact numeric types with binary precision in bits and a scale of 0 (zero) are collectively referred to as (signed or unsigned) *integer types*. Values of (signed or unsigned) integer types are referred to as (signed or unsigned) *integer numbers*, or, alternatively as (signed or unsigned) *integers*.

ISO/IEC 39075:2024(en)
4.16 Predefined value types

- Exact numeric types with decimal precision and scale in digits are referred to as *decimal types*. Values of decimal types are collectively referred to as *decimal numbers*.
- The value types specified with the leading keywords FLOAT, FLOAT16, FLOAT32, FLOAT64, FLOAT128, FLOAT256, REAL, DOUBLE, and DOUBLE PRECISION are collectively referred to as *approximate numeric types* and the values of approximate numeric types are known as *approximate numbers*, or, alternatively, as *floating point numbers*. The primary base type of all approximate numeric types is the approximate numeric type base type.
- *Temporal instant types* and *temporal duration types* are collectively referred to as temporal types.
- The value types specified with the leading keywords ZONED DATETIME, LOCAL DATETIME, DATE, ZONED TIME, and LOCAL TIME are collectively referred to as temporal instant types. The primary base type of all temporal instant types is the temporal instant type base type.
- The value types specified by DURATION(YEAR TO MONTH) and DURATION(DAY TO SECOND) are collectively referred to as temporal duration types. The primary base type of all temporal duration types is the temporal duration type base type.
- Binding table reference value types, graph reference value types, and graph element reference value types are collectively referred to as reference value types. Values of reference value types are referred to as reference values.
- The reference value types specified with the leading keywords BINDING TABLE and TABLE are collectively referred to as binding table reference values. The primary base type of all binding table reference value types is the binding table reference type base type.
- The reference value types specified with the leading keywords ANY PROPERTY GRAPH, PROPERTY GRAPH, ANY GRAPH, and GRAPH are collectively referred to as graph reference values, or, alternatively, as property graph reference values. The primary base type of all graph reference value types is the graph reference type base type.
- Node reference value types and edge reference value types are collectively referred to as graph element reference value types. Values of graph element reference value types are referred to as graph element reference values.
- The reference value types specified with the leading keywords ANY NODE, NODE, ANY VERTEX, and VERTEX are collectively referred to as node reference values, or, alternatively, as vertex reference values. The primary base type of all node reference value types is the node reference type base type.
- The reference value types specified with the leading keywords ANY EDGE, EDGE, ANY RELATIONSHIP, and RELATIONSHIP are collectively referred to as edge reference values, or, alternatively, as relationship reference values. The primary base type of all edge reference value types is the edge reference type base type.
- The value types specified by NULL, NULL NOT NULL, and NOTHING are collectively referred to as immaterial value types. The primary base type of all immaterial value types is the immaterial value type base type.

4.16.2 Boolean types

The material values of a *Boolean type* are the distinct truth values *True* or *False*. The truth value *Unknown* is represented by the null value.

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

Every Boolean type is described by its Boolean data type descriptor. A Boolean data type descriptor comprises:

- The declared name of the primary base type of all Boolean types (BOOLEAN DATA).
- The preferred name of the Boolean type BOOLEAN.
- The indication of whether the type includes the null value.

A GQL-implementation regards certain <boolean type>s as equivalent, if they have the same indication regarding the inclusion of the null value, as permitted by the Syntax Rules of [Subclause 18.9](#), “<value type>”. When two or more <boolean type>s are equivalent, the GQL-implementation chooses one of these equivalent <boolean type>s as the normal form representing that equivalence class of <boolean type>s.

A truth value is assignable to a site of at least the material Boolean type. The values *True* and *False* can be assigned to any site having at least material Boolean type; assignment of *Unknown*, or the null value, is subject to the nullability of the target.

NOTE 130 — See [Subclause 4.17.4](#), “Nullability”.

Any two Boolean values are essentially comparable values.

4.16.3 Character string types

4.16.3.1 Introduction to character strings

A material value of a *character string type* is a character string. A character string is a possibly empty sequence of characters drawn from the Universal Character Set repertoire specified by [The Unicode® Standard](#).

A character string has a length that is the number of characters in the sequence. The length is 0 (zero) or a positive integer. The maximum length of a character string is implementation-defined ([IL013](#)) but shall be greater than or equal to $2^{14} - 1 = 16383$ characters.

A GQL-implementation may assume that all character strings are normalized in one of Normalization Form C (NFC), Normalization Form D (NFD), Normalization Form KC (NFKC), or Normalization Form KD (NFKD). These normalization forms shall be in accordance with [Unicode Standard Annex #15](#). <normalized predicate> can be used to verify the normalization form to which a particular character string conforms. Applications can also use <normalize function> to enforce a particular <normal form>. With the exception of <normalize function> and <normalized predicate>, the result of any operation on an unnormalized character string is implementation-defined ([IA003](#)).

A GQL-implementation may determine two character strings to be *visually confusable* with each other using an implementation-defined ([IW014](#)) mechanism. A character string shall never be determined to be visually confusable with itself.

NOTE 131 — This definition deliberately includes the possibility of never determining two character strings to be visually confusable with each other.

NOTE 132 — A major source of visually confusable character strings are homographs. implementers are advised to consult [Unicode Standard Annex #36](#) for further information on visually confusable character strings and homographs.

Every character string type is described by its character string data type descriptor. A character string data type descriptor comprises:

- The declared name of the primary base type of all character string types (STRING DATA).
- The preferred name of the character string type (implementation-defined ([ID023](#)) choice of STRING, CHAR, or VARCHAR).
- The minimum length in characters of the character string type.

- The maximum length in characters of the character string type.
- The indication of whether the type includes the null value.

The minimum length in characters of a character string type is a non-negative integer.

The maximum length in characters of a character string type is a positive integer.

Character string types where the minimum length is equal to the maximum length are *fixed-length character string types*. The minimum length and the maximum length of a fixed-length character string type are simply referred to as the length of that character string type. Character string types where the minimum length is less than the maximum length are *variable-length character string types*.

The preferred name of character string types is determined by an implementation-defined (ID023) choice of either STRING as the preferred name of all character string types or, alternatively, CHAR as the preferred name of all fixed-length character string types and VARCHAR as the preferred name of all variable-length character string types.

A GQL-implementation regards certain <character string type>s as equivalent, if they have the same minimum length, maximum length, and indication regarding the inclusion of the null value, as permitted by the Syntax Rules of Subclause 18.9, “<value type>”. When two or more <character string type>s are equivalent, the GQL-implementation chooses one of these equivalent <character string type>s as the normal form representing that equivalence class of <character string type>s.

Any two character string values are essentially comparable values.

A character string *CS* is assignable to a site that is at least of some material character string type *CST* if and only if the length of *CS* is both greater than or equal to the minimum length and less than or equal to the maximum length of *CST*.

NOTE 133 — Assuming that the declared type of the site is a closed dynamic union type *DUT*, it holds that *CS* is assignable to the site if and only if the length of *CS* is both greater than or equal to the minimum length as well as less than or equal to the maximum length of at least one character string type included in the component types of *DUT*.

If evaluation of a <cast specification> where the source and target are both character strings would result in the loss of non-<truncating whitespace> characters due to truncation, then a warning condition is raised. If a store assignment would result in the loss of non-<truncating whitespace> characters due to truncation, then an exception condition is raised.

4.16.3.2 Collations

Subclause 3.8, “Value terms and definitions”, defines a collation as a process by which, given two strings, it is determined whether the first one is less than, equal to, or greater than the second one.

The GQL-implementation defines exactly one *default collation* that is used in the comparison of character strings and which is defined as the implementation-defined (ID022) choice of one of the following:

- UCS_BASIC, the collation in which the ordering is determined entirely by the Unicode scalar values of the characters in the character strings being sorted.

NOTE 134 — The Unicode scalar value of a character is its code point treated as an unsigned integer.

- UNICODE, the collation in which the ordering is determined by applying the Unicode Collation Algorithm with the Default Unicode Collation Element Table, in accordance with Unicode Technical Standard #10.
- An implementation-defined (ID022) custom collation provided by the GQL-implementation.

4.16.4 Byte string types

A material value of a *byte string type* is a byte string. A byte string is a possibly empty sequence of bytes (octets).

A byte string has a length that is the number of bytes in the sequence. The length is 0 (zero) or a positive integer. The maximum length of a byte string is implementation-defined (IL013) but shall be greater than or equal to $2^{16}-2=65534$ bytes.

Every byte string type is described by its byte string data type descriptor. A byte string data type descriptor comprises:

- The declared name of the primary base type of all byte string types (BINARY DATA).
- The preferred name of the byte string type (implementation-defined (ID023) choice of BINARY, BYTES, or VARBINARY).
- The minimum length in bytes of the byte string type.
- The maximum length in bytes of the byte string type.
- The indication of whether the byte string type includes the null value.

The minimum length in bytes of a byte string type is a non-negative integer.

The maximum length in bytes of a byte string type is a positive integer.

Byte string types where the minimum length is equal to the maximum length are *fixed-length byte string types*. The minimum length and the maximum length of a fixed-length byte string type are simply referred to as the length of that byte string type. Byte string types where the minimum length is less than the maximum length are *variable-length byte string types*.

The preferred name of byte string types is determined by an implementation-defined (ID023) choice of either BYTES as the preferred name of all byte string types or, alternatively, BINARY as the preferred name of all fixed-length byte string types and VARBINARY as the preferred name of all variable-length byte string types.

A GQL-implementation regards certain <byte string type>s as equivalent, if they have the same minimum length, maximum length and indication regarding the inclusion of the null value, as permitted by Subclause 18.9, "<value type>". When two or more <byte string type>s are equivalent, the GQL-implementation chooses one of these equivalent <byte string type>s as the normal form representing that equivalence class of <byte string type>s.

Any two byte string values are essentially comparable values.

A byte string *BS* is assignable to a site that is at least of some material byte string type *BST* if and only if the length of *BS* is both greater than or equal to the minimum length and less than or equal to the maximum length of *BST*.

NOTE 135 — Assuming that the declared type of the site is a closed dynamic union type *DUT*, it holds that *BS* is assignable to the site if and only if the length of *BS* is both greater than or equal to the minimum length as well as less than or equal to the maximum length of at least one byte string type included in the component types of *DUT*.

If evaluation of a <cast specification> would result in the loss of bytes due to truncation, then a warning condition is raised. If a store assignment would result in the loss of bytes due to truncation, then an exception condition is raised.

4.16.5 Numeric types

4.16.5.1 Introduction to numbers

GQL supports two classes of numeric data:

- Exact numeric data.
- Approximate numeric data.

Exact numeric data is either signed or unsigned. Signed numeric data is either non-negative (positive or zero) or negative. Unsigned numeric data is always non-negative. Approximate numeric data is always signed.

Numeric types are either *binary* or *decimal*, i.e., either their material values (numbers) are specified in binary terms with a radix of 2 or, respectively, are specified in decimal terms with a radix of 10. Signed binary exact numeric types are two's complement integers. Binary exact numeric types do not specify a scale factor. Decimal exact numeric types may specify a scale factor. Approximate numeric types are always binary numeric types and may specify a scale factor.

Every *numeric type* is described by a numeric data type descriptor. A numeric data type descriptor comprises:

- The declared name of the primary base type of the numeric type (EXACT NUMERIC DATA for exact numeric types and FLOAT NUMERIC DATA for approximate numeric types).
- The preferred name of the specific numeric type, which is the declared name of the normal form of the numeric type.
- The indication of whether numbers of the numeric type are specified in binary or decimal terms. The radix of binary exact numbers is 2, the radix of decimal exact numbers is 10, and the radix of approximate numbers is always 2.
- The precision of the numeric type, which is a positive integer.
- The scale of the numeric type, which is a non-negative integer.
- The indication of whether the type includes the null value.

A GQL-implementation is permitted to regard two <exact numeric type>s as equivalent, if they have the same precision, scale, radix, and indication regarding the inclusion of the null value, as permitted by the Syntax Rules of [Subclause 18.9](#), "<value type>". When two or more <exact numeric type>s are equivalent, the GQL-implementation chooses one of these equivalent <exact numeric type>s as the normal form representing that equivalence class of <exact numeric type>s. The normal form determines the preferred name of the exact numeric type in the numeric data type descriptor.

Similarly, a GQL-implementation is permitted to regard two <approximate numeric type>s as equivalent, if they have the same precision, scale, and indication regarding the inclusion of the null value, as permitted by the Syntax Rules of [Subclause 18.9](#), "<value type>". When two or more <approximate numeric type>s are equivalent, the GQL-implementation chooses a normal form to represent each equivalence class of <approximate numeric type>s. The normal form determines the preferred name of the approximate numeric type in the numeric data type descriptor.

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

4.16.5.2 Characteristics of numbers

A numeric type has a precision P and a scale S . P is a positive integer that determines the number of significant digits of numbers of the numeric type in a particular radix R , where R is either 2 or 10. S is a non-negative integer such that $S \leq P$. Every value of a signed binary exact numeric type of scale S is of the form

ISO/IEC 39075:2024(en)
4.16 Predefined value types

$N \times 10^{-S}$, where N is an integer such that $-(2^P) \leq N < 2^P$. Every value of an unsigned binary exact numeric type of scale S is representable in the form $N \times R^{-S}$, where N is an integer such that 0 (zero) $\leq N < R^P$. Every value of a signed decimal exact numeric type of scale S is representable in the form $N \times 10^{-S}$, where N is an integer such that $-(10^P) < N < 10^P$.

NOTE 136 — Not every value in that range is necessarily a value of the type in question.

Exact numeric values are effectively specified by an integral part and a fractional part. The integral part is a signed numeric value that specifies the sign of the numeric value and the digits before the radix point of the numeric value and the fractional part is a non-negative integer that specifies the digits after the radix point of the numeric value. The value of an exact number is determined by application of the normal mathematical interpretation of positional notation to the concatenation of the sign and the digits before the radix point specified by the integral part, followed by the radix point and the digits after the radix point specified by the fractional part.

Approximate numeric values are effectively specified by 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. Approximate numeric values have a precision and a scale. The precision of approximate numeric values is a positive integer that specifies the number of significant binary digits in the mantissa. In this document, the sign of the mantissa is not considered as a significant binary digit for the purpose of determining the precision of an approximate numeric value. The scale of approximate numeric values is its exponent size, i.e., a signed integer that specifies the number of significant binary digits of the exponent. In this document, the sign of the exponent is not considered as a significant binary digit for the purpose of determining the scale of an approximate numeric value. The value of an approximate number is the mantissa multiplied by a factor determined by the exponent.

NOTE 137 — A GQL-implementation can choose an internal representation for approximate numeric values that implicitly encodes leading bits instead of physically storing them. Any such implied bits are still part of the mantissa of any value represented using such an encoding.

An <exact numeric literal> *ENL* comprises either an <unsigned decimal in scientific notation> followed by an <exact number suffix>, an <unsigned decimal in common notation> optionally followed by an <exact number suffix>, an <unsigned decimal integer> followed by an <exact number suffix>, or an <unsigned integer>. There is an <exact numeric literal> *ENL2* that does not simply contain an <unsigned decimal in scientific notation> such that *ENL2* is equivalent to *ENL* and *ENL2* specifies the same exact number as *ENL*. The declared type of *ENL* is an exact numeric type.

An <approximate numeric literal> *ANL* comprises either an <unsigned decimal in scientific notation> optionally followed by an <approximate number suffix>, an <unsigned decimal in common notation> followed by an <approximate number suffix>, or an <unsigned decimal integer> followed by an <approximate number suffix>. There is an <unsigned decimal in scientific notation> *UDSN* that is equivalent to *ANL* and that specifies the same approximate number as *ANL*. The declared type of *ANL* is an approximate numeric type. If M is the value of the <mantissa> and E is the value of the <exponent> of *UDSN*, then $M \times 10^E$ is the *apparent value* of *ANL*. If the declared type of *ANL* is an approximate numeric type, then the actual value of *ANL* is approximately the apparent value of *ANL*, according to implementation-defined (IA004) rules.

An <unsigned decimal in scientific notation> comprises a <mantissa> that is an <unsigned decimal integer> or an <unsigned decimal in common notation> of the value, the letter “E” or “e”, and an <exponent> that is a <signed decimal integer> that specifies an unsigned number by specifying a mantissa and an exponent. An <unsigned decimal in common notation> comprises either an <unsigned decimal integer>, a <period> followed by an <unsigned decimal integer>, or an <unsigned decimal integer> followed by a <period> and an optional <unsigned decimal integer> that specifies an unsigned number by specifying the sequences of digits before and after the decimal point. An <unsigned integer> comprises a sequence of digits that specifies an unsigned integer in decimal, hexadecimal, octal, or binary terms.

Any two numbers are essentially comparable values.

ISO/IEC 39075:2024(en)
4.16 Predefined value types

A number is assignable to a site of at least material exact numeric type or the material approximate numeric type. If an assignment of some number would result in a loss of its most significant digit, an exception condition is raised. If least significant digits are lost, it is implementation-defined (IA021) if an exception condition is raised or if truncation or rounding occurs. In the latter case, the choice of whether to truncate or round is implementation-defined (IA005). The rules for arithmetic are specified in Subclause 20.21, "<numeric value expression>".

Whenever a 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 (IA005).

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 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 (IA006) 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 (IA007) 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 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 (IA006) which one is taken.

Whenever a 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 (IA010) limits, except as provided for in Subclause 20.21, "<numeric value expression>".

4.16.5.3 Binary exact numeric types

The signed binary exact numeric types are:

- The *signed 8-bit integer type* specified by SIGNED INTEGER8 (alternatively: INTEGER8, INT8) with precision 7 and with scale 0 (zero).
- The *signed 16-bit integer type* specified by SIGNED INTEGER16 (alternatively: INTEGER16, INT16) with precision 15 and with scale 0 (zero).
- The *signed 32-bit integer type* specified by SIGNED INTEGER32 (alternatively: INTEGER32, INT32) with precision 31 and with scale 0 (zero).
- The *signed 64-bit integer type* specified by SIGNED INTEGER64 (alternatively: INTEGER64, INT64) with precision 63 and with scale 0 (zero).
- The *signed 128-bit integer type* specified by SIGNED INTEGER128 (alternatively: INTEGER128, INT128) with precision 127 and with scale 0 (zero).
- The *signed 256-bit integer type* specified by SIGNED INTEGER256 (alternatively: INTEGER256, INT256) with precision 255 and with scale 0 (zero).

ISO/IEC 39075:2024(en)
4.16 Predefined value types

- The *signed regular integer type* specified by SIGNED INTEGER (alternatively: INTEGER, INT) with implementation-defined (ID028) precision greater than or equal to the precision of the signed regular integer type and with scale zero (0).
- The *signed small integer type* specified by SIGNED SMALL INTEGER (alternatively: SMALL INTEGER, SMALLINT) with implementation-defined (ID028) precision less than or equal to the precision of the signed regular integer type and with scale zero (0).
- The *signed big integer type* specified by SIGNED BIG INTEGER (alternatively: BIG INTEGER, BIGINT) with implementation-defined (ID028) precision greater than or equal to the precision of the signed regular integer type and with scale zero (0).
- The *signed user-specified integer types* specified by SIGNED INTEGER(p) (alternatively: INTEGER(p), INT(p)) with implementation-defined (ID028) precision greater than or equal to p and with scale zero (0).

The unsigned binary exact numeric types are:

- The *unsigned 8-bit integer type* specified by UNSIGNED INTEGER8 (alternatively: UINT8) with precision 8 and with scale 0 (zero).
- The *unsigned 16-bit integer type* specified by UNSIGNED INTEGER16 (alternatively: UINT16) with precision 16 and with scale 0 (zero).
- The *unsigned 32-bit integer type* specified by UNSIGNED INTEGER32 (alternatively: UINT32) with precision 32 and with scale 0 (zero).
- The *unsigned 64-bit integer type* specified by UNSIGNED INTEGER64 (alternatively: UINT64) with precision 64 and with scale 0 (zero).
- The *unsigned 128-bit integer type* specified by UNSIGNED INTEGER128 (alternatively: UINT128) with precision 128 and with scale 0 (zero).
- The *unsigned 256-bit integer type* specified by UNSIGNED INTEGER256 (alternatively: UINT256) with precision 256 and with scale 0 (zero).
- The *unsigned regular integer type* specified by UNSIGNED INTEGER (alternatively: UINT) with the same precision as the precision of the unsigned 32-bit integer type and with scale zero (0).
- The *unsigned small integer type* specified by UNSIGNED SMALL INTEGER (alternatively: USMALLINT) with implementation-defined (ID028) precision less than or equal to the precision of the unsigned regular integer type and with scale zero (0).
- The *unsigned big integer type* specified by UNSIGNED BIG INTEGER (alternatively: UBIGINT) with implementation-defined (ID028) precision greater than or equal to the precision of the unsigned regular integer type and with scale zero (0).
- The *unsigned user-specified integer types* specified by UNSIGNED INTEGER(p) (alternatively: UINT(p)) with implementation-defined (ID028) precision greater than or equal to p and with scale zero (0).

4.16.5.4 Decimal exact numeric types

The decimal exact numeric types are:

- The *regular decimal exact numeric type* specified by DECIMAL (alternatively: DEC) with implementation-defined (ID034) precision greater than or equal to the precision of the signed regular integer type and with scale 0 (zero).
- The *user-specified decimal exact numeric types* specified by DECIMAL(p) (alternatively: DEC(p)) with implementation-defined (ID034) precision greater than or equal to p and with scale 0 (zero).

- The *user-specified decimal exact numeric types* specified by $\text{DECIMAL}(p, s)$ (alternatively: $\text{DEC}(p, s)$) with implementation-defined (ID034) precision greater than or equal to p and with implementation-defined (ID037) scale of s .

4.16.5.5 Approximate numeric types

The approximate numeric types are:

NOTE 138 — The precision and scale of an approximate numeric type specify the number of most significant binary digits except for the sign of the mantissa and, respectively, the exponent of approximate numbers of the type.

- The *16-bit approximate numeric type* specified by FLOAT16 with precision 10 and with scale 4.
NOTE 139 — The material value space of the 16-bit approximate numeric type FLOAT16 is defined to be compatible with the binary16 interchange format of [IEEE Std 754:2019](#).
- The *32-bit approximate numeric type* specified by FLOAT32 with precision 23 and with scale 7.
NOTE 140 — The material value space of the 32-bit approximate numeric type FLOAT32 is defined to be compatible with the binary32 interchange format of [IEEE Std 754:2019](#).
- The *64-bit approximate numeric type* specified by FLOAT64 with precision 52 and with scale 10.
NOTE 141 — The material value space of the 64-bit approximate numeric type FLOAT64 is defined to be compatible with the binary64 interchange format of [IEEE Std 754:2019](#).
- The *128-bit approximate numeric type* specified by FLOAT128 with precision 112 and with scale 14.
NOTE 142 — The material value space of the 128-bit approximate numeric type FLOAT128 is defined to be compatible with the binary128 interchange format of [IEEE Std 754:2019](#).
- The *256-bit approximate numeric type* FLOAT256 with precision 236 and with scale 18.
NOTE 143 — The material value space of the 256-bit approximate numeric type specified by FLOAT256 is defined to be compatible with the binary256 interchange format of [IEEE Std 754:2019](#).
- The *regular approximate numeric type* specified by FLOAT with implementation-defined (ID037) precision greater than or equal to 23 and with implementation-defined (ID037) scale greater than or equal to 7.
- The *real approximate numeric type* specified by REAL with implementation-defined (ID037) precision less than or equal to the precision of the regular approximate numeric type and with implementation-defined (ID037) scale.
- The *double approximate numeric type* specified by DOUBLE (alternatively: DOUBLE PRECISION) with implementation-defined (ID037) precision greater than or equal to the precision of the regular approximate numeric type and with implementation-defined (ID037) scale.
- The *user-specified approximate numeric types* specified by $\text{FLOAT}(p)$ with implementation-defined (ID037) precision greater than or equal to p and with implementation-defined (ID037) scale.
- The user-specified approximate numeric types specified by $\text{FLOAT}(p, s)$ with implementation-defined (ID037) precision greater than or equal to p and with implementation-defined (ID037) scale greater than or equal to s .

If a GQL-implementation supports Feature GA01, “IEEE 754 floating point operations”, a <numeric value expression> s on approximate numeric types that would otherwise result in exceptions may return additional values. Any additional value returned shall be one defined by [IEEE Std 754:2019](#). It is implementation-defined (IA025) what the effect of these additional values have on the rest of GQL. However, to the extent that this document and [IEEE Std 754:2019](#) provide similar operations, it is recommended that the GQL-implementation defines the behavior of such operations on any such additionally introduced values to be in accordance with [IEEE Std 754:2019](#).

4.16.6 Temporal types

4.16.6.1 Introduction to temporal data

There are two classes of temporal data:

- Temporal instant data.
- Temporal duration data.

The specifications of temporal types for temporal data reference the formats and operations that are specified in [ISO 8601-1:2019](#) and [ISO 8601-2:2019](#) but also support the temporal literal formats specified in [ISO/IEC 9075-2:2023](#).

The time scale used for temporal instant data shall conform to [UTC-SLS](#).

4.16.6.2 Temporal instant types

The temporal instant types are:

- ZONED DATETIME

A material value of ZONED DATETIME is called a *zoned datetime*. A zoned datetime represents a temporal instant capturing the date, the time, and the time zone displacement.

NOTE 144 — Equivalent to `TIMESTAMP WITH TIME ZONE` with nanosecond precision in SQL.

- LOCAL DATETIME

A material value of LOCAL DATETIME is called a *local datetime*. A local datetime represents a temporal instant capturing the date and the time, but not the time zone displacement.

NOTE 145 — Equivalent to `TIMESTAMP WITHOUT TIME ZONE` with nanosecond precision in SQL.

- DATE

A material value of DATE is called a *date*. A date represents a temporal instant capturing the date, but neither the time, nor the time zone displacement.

NOTE 146 — Equivalent to `DATE` in SQL.

- ZONED TIME

A material value of ZONED TIME is called a *zoned time*. A zoned time represents a temporal instant capturing the time of day and the time zone displacement, but not the date.

NOTE 147 — Equivalent to `TIME WITH TIME ZONE` with nanosecond precision in SQL.

- LOCAL TIME

A material value of LOCAL TIME is called a *local time*. A local time represents a temporal instant capturing the time of day, but neither the date, nor the time zone displacement.

NOTE 148 — Equivalent to `TIME WITHOUT TIME ZONE` with nanosecond precision in SQL.

A *datetime* is either a zoned datetime or a local datetime and a *time* is either a zoned time or a local time.

Every temporal instant type is described by a temporal instant data type descriptor. A temporal instant data type descriptor comprises:

- The declared name of the primary base type of all temporal instant types (`TEMPORAL INSTANT DATA`).
- The preferred name of the specific temporal instant type, which is the declared name of the normal form of the temporal instant type.

ISO/IEC 39075:2024(en)
4.16 Predefined value types

- The indication of whether the temporal instant values of the type capture the date.
- The indication of whether the temporal instant values of the type capture the time of day.
- The indication of whether the temporal instant values of the type include a time zone displacement.
- The indication of whether the type includes the null value.

The format of temporal instant literals shall be either in accordance with [ISO 8601-1:2019](#) and [ISO 8601-2:2019](#) or in accordance with [ISO/IEC 9075-2:2023](#).

NOTE 149 — See [Subclause 21.2](#), “<literal>”.

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 *Universal Coordinated Time (UTC)* plus the *time zone displacement*. The time zone displacement is constant throughout a time zone, and can change at the beginning and end of Summer Time, where applicable.

A time zone is represented by the time zone displacement between the local time and UTC. Time zone displacement is defined in [ISO 8601-1:2019](#) where it is referred to as time shift. The representation of a time zone displacement is effectively that of an ISO 8601 time shift.

A datetime is assignable to a site of at least the material zoned datetime type or the material local datetime type. A date is assignable to a site of at the material least date type. A time is assignable to a site of at least the material zoned time type or the material local time type.

For the convenience of users, whenever a zoned datetime is to be implicitly derived from a local datetime (for example, in a simple assignment operation), GQL assumes the local datetime to be given in the current time zone displacement *CTZD* (after implicitly casting *CTZD* as a day and time-based duration), subtracts *CTZD* from it to give UTC, and associates *CTZD* to obtain the zoned datetime result.

NOTE 150 — See [Subclause 4.16.6.3](#), “Temporal duration types”.

Conversely, whenever a local datetime is to be implicitly derived from a zoned datetime, GQL assumes the zoned datetime to be UTC, adds the time zone displacement to it to give local time, and obtains the local datetime result.

Temporal instants of the same most specific static value types are essentially comparable values.

A GQL-implementation regards certain <temporal instant type>s as equivalent, if they have the same indications regarding whether their values capture the date, whether their values capture the time of day, whether their values include a time zone displacement, and regarding the inclusion of the null value, as permitted by the Syntax Rules of [Subclause 18.9](#), “<value type>”. When two or more <temporal instant type>s are equivalent, the GQL-implementation chooses one of these equivalent <temporal instant type>s as the normal form representing that equivalence class of <temporal instant type>s. The normal form determines the preferred name of the temporal instant type in the temporal instant data type descriptor.

4.16.6.3 Temporal duration types

The material values of temporal duration types are temporal duration values. A temporal duration captures a time difference, i.e., a temporal amount in specified units that represents the difference in time between two temporal instants. It only captures the amount of time between two temporal instants; it does not capture a start time and an end time. A temporal duration value can be positive, zero, or negative.

Temporal durations capture time differences as exact numeric multiples of a few different temporal duration units. Every temporal duration only uses the temporal duration units from exactly one of the following *temporal duration unit groups*:

- The *year and month-based duration unit group* which comprises the following temporal duration units: Years and Months.

ISO/IEC 39075:2024(en)
4.16 Predefined value types

- The *day and time-based duration unit group* which comprises the following temporal duration units: Days, Hours, Minutes, Seconds, and Subseconds (Milliseconds, Microseconds, Nanoseconds).

NOTE 151 — These temporal duration unit groups are identical to the year-month and day-time interval classes in SQL.

A *year and month-based duration* is a temporal duration specified using temporal duration units from the year and month-based duration unit group only. Similarly, a *day and time-based duration* is a temporal duration specified using temporal duration units from the day and time-based duration unit group only.

Conversion between temporal duration units is only possible between temporal duration units from the same temporal duration unit group but not between temporal duration units from different temporal duration unit groups (other than through applying a temporal duration to a point in time).

Every temporal duration type is described by a temporal duration data type descriptor. A temporal duration data type descriptor comprises:

- The declared name of the primary base type of all temporal duration types (TEMPORAL DURATION DATA).
- The temporal duration unit group of temporal durations of the type (one of: year and month-based duration unit group or day and time-based duration unit group).
- The indication of whether the type includes the null value.

The format of temporal duration literals shall be either in accordance with ISO 8601-1:2019 and ISO 8601-2:2019 or in accordance with ISO/IEC 9075-2:2023.

NOTE 152 — See Subclause 21.2, “<literal>”.

A *year and month-based duration type* is a temporal duration type that specifies the year and month-based duration unit group. Similarly, a *day and time-based duration type* is a temporal duration type that specifies the day and time-based duration unit group.

A year and month-based duration is assignable to a site of at least the material year and month-based duration type. Similarly, a day and time-based duration is assignable to a site of at least the material day and time-based duration type.

Temporal duration values whose specified units are of the same temporal duration unit group are essentially comparable values.

4.16.6.4 Operators involving values of temporal types

Table 1, “Valid operators involving values of temporal types”, specifies the declared types of arithmetic expressions involving temporal instant, temporal duration, and number operands.

NOTE 153 — Regarding the subtraction of temporal instants, the difference between two temporal instants given as *TI1* and *TI2* and assumed to be specifiable by the <datetime value expression>s *DVE1* and *DVE2*, respectively, is the result of DURATION_BETWEEN(*DVE1*, *DVE2*). See Subclause 20.28, “<duration value expression>”.

Table 1 — Valid operators involving values of temporal types

Operand 1	Operator	Operand 2	Result
temporal instant	-	temporal instant	temporal duration
temporal instant	+ or -	temporal duration	temporal instant
temporal duration	+	temporal instant	temporal instant

ISO/IEC 39075:2024(en)
4.16 Predefined value types

Operand 1	Operator	Operand 2	Result
temporal duration	+ or -	temporal duration	temporal duration
temporal duration	* or /	number	temporal duration
number	*	temporal duration	temporal duration

Arithmetic operations involving values of a temporal instant type or a temporal duration type obey the rules associated with dates and times and yield valid temporal instant or temporal duration results according to the Gregorian calendar as defined in [ISO 8601-1:2019](#).

The details of subtracting one temporal instant from another temporal instant are specified in this document. The details of other operations involving temporal types are specified in [ISO 8601-2:2019](#).

Operations involving values of a temporal instant type require that the temporal instants be essentially comparable values. Operations involving values of a temporal duration type require that the temporal durations be essentially comparable values.

Operations involving a temporal instant and a temporal duration preserve the time zone displacement of the temporal instant operand. If the temporal instant operand does not include a time zone displacement, then the result has no time zone displacement.

Temporal instants and temporal durations that contain seconds have an implementation-defined ([IL024](#)) maximum seconds precision value that is not less than 6.

4.16.7 Reference value types

A material value of a reference value type is a reference value. A reference value is a globally resolved reference that effectively represents a reference to some globally identifiable object (its referent) by opaquely encapsulating its referent's global object identifier.

The GQL language supports the following kinds of reference values:

- Graph reference values whose referents are graphs.
- Binding table reference values whose referents are binding tables.
- Node reference values whose referents are nodes.
- Edge reference values whose referents are edges.

Every reference value type is described by a reference value data type descriptor. A reference value data type descriptor comprises:

- The *reference base type name* of the reference value type. This name always ends with the word REFERENCE.
- The *object base type name* of the reference value type. This name is the name of the common base type of the GQL-object types of all possible referents of the reference values of the reference value type.

NOTE 154 — In this document, the reference base type name and the object base type name of a reference value type always have the same base type name prefix.

- The preferred name of the reference value type.
- The optional *constraining GQL-object type* of all possible referents of the reference values of the reference value type. The name of the primary base type of this GQL-object type is the object base type name.

NOTE 155 — In this document, the preferred name of the reference value type and the constraining GQL-object type (if any) are the same.

- The indication of whether the type contains the null value.

A reference value whose referent is *O* is assignable to a site that is at least of some material reference value type *RVT* whose object base type is *OBT* if the primary base type of *O* is *OBT* and, given that *RVT* has a constraining GQL-object type *COT*, it holds that *O* is assignable to *COT*.

4.16.8 Immaterial value types: null type and empty type

The immaterial value types defined in this document are the *null type* and the *empty type*. The null type is the (nullable) value type comprising the (single) null value shared by all nullable types. The empty type is the material variant of the null type. Hence, the empty type comprises no values (is the empty set) and a <value expression> whose declared type is the empty type cannot be evaluated. In this document, the empty type is primarily used as the declared type of omitted results but it may also occur as a column type of an empty binding table in the GQL-catalog. Supported property value types are non-immaterial value types by definition.

NOTE 156 — See Subclause 4.4.5, “Material values and the null value”.

The null type is a subtype of every nullable value type and the empty type is a subtype of every value type.

The immaterial value types are only provided by GQL-implementations that support the Feature GV70, “Immaterial value types”. In such GQL-implementations, the null type provides a portable most specific value type for various values (e.g., certain constructed values such as empty list values or the null value).

Every immaterial value type is described by its immaterial data type descriptor. An immaterial data type descriptor comprises:

- The declared name of the primary base type of all immaterial types (NULL DATA).
- The preferred name of the immaterial type (NULL for the null type and NOTHING for the empty type).
- The indication of whether the type includes the null value.

A GQL-implementation regards certain <immaterial value type>s as equivalent, if they have the same indication regarding the inclusion of the null value, as permitted by the Syntax Rules of Subclause 18.9, “<value type>”. When two or more <immaterial value type>s are equivalent, the GQL-implementation chooses one of these equivalent <immaterial value type>s as the normal form representing that equivalence class of <immaterial value type>s.

For every value *V* it holds that the null value and *V* are essentially comparable values.

A null value is assignable to a site of at least a nullable type. The instance occupying a site whose declared type is the empty type cannot be assigned nor determined.

4.17 Sites

4.17.1 General description of sites

A site is a place occupied by an instance of a specified data type. Every site has a defined degree of persistence, independent of its declared type (if it has a declared type). A site that exists until deliberately destroyed is said to be persistent. A site that necessarily ceases to exist on completion of a statement, at the end of a GQL-transaction, or at the end of a GQL-session is said to be temporary. A site that exists only for as long as necessary to hold a GQL-object or a value during the execution of an operation is said to be *transient*.

4.17.2 Static and dynamic sites

Every site is either a *static site* or a *dynamic site*. A static site is a site occupied by an instance known or determined during the application of Syntax Rules. A dynamic site is a site occupied by instances known or determined during the application of General Rules.

The GQL language defines and interacts with the following kinds of static sites:

- Session parameters.
- Request parameters.
- Working schemas.
- The values of <value specification>s.
- Sites that are occupied by primary objects in the GQL-catalog.

Every other site is a dynamic site.

4.17.3 Assignment and store assignment

The value at a site is set by the operation of assignment. Assignment initializes or replaces the value at a site T (known as the target) whose declared type is DT with a new (possibly different) value S (known as the source).

NOTE 157 — GQL-objects are inserted or removed (e.g., into the GQL-schema or other GQL-objects) but never assigned. However, reference values to GQL-objects are subject to provisions regarding assignment.

There are two kinds of assignment:

- Regular assignment, which is frequently indicated by using the phrases “ T is assigned to S ”, “the value of T is assigned to S ”, or “the value of S assigned from T ”.
- Store assignment, which is always performed by explicitly calling the General Rules of [Subclause 22.10, “Store assignment”](#).

By default, assignment is regular assignment.

If S is included in DT , then S is assigned to T as is. If S is not included in DT but the result $S1$ of converting S to DT using relevant type conversions is defined, then $S1$ is assigned to T (instead of S). Otherwise, S is not assignable to T . In particular, the null value is only assignable to a site whose declared type is nullable.

The relevant type conversions used depend on the kind of assignment that is being performed. (Regular) assignment applies type conversions that are specified as part of provisions on the assignment of values of individual value types in [Subclause 4.14, “Dynamic union types”](#), [Subclause 4.15, “Constructed value types”](#), and [Subclause 4.16, “Predefined value types”](#), while store assignment applies the type conversions specified in the General Rules of [Subclause 22.10, “Store assignment”](#).

4.17.4 Nullability

4.17.4.1 Introduction to nullability

Every site has a *nullability* characteristic that indicates whether the site may be occupied by the null value (is *possibly nullable*) or not (is *known not nullable*). A nullability $N1$ is assignment-aligned to a nullability $N2$ if and only if either $N1$ and $N2$ are the same nullability or it holds that $N1$ is known not nullable and $N2$ is possibly nullable. In this document, the nullability of a site S is defined as follows. If the data type descriptor of the declared type DT of S includes an indication IND regarding the inclusion of the null value in DT , then the nullability of S is as specified by IND ; otherwise, S is known not nullable.

4.17.4.2 Nullability requirements

If a Syntax Rule requires the declared type *DT* of a site *S* to be of some required type *RT*, then the nullability of *DT* needs to be assignment-aligned with the nullability of *RT*.

NOTE 158 — For example, this is the case if *DT* is the material variant of *RT*. This is a consequence of subtyping since every material type is a subtype of its nullable variant.

4.17.4.3 Nullability inference

A type stated in this document without explicitly specifying its nullability implies its nullable variant by default and accordingly all sites of such a type are possibly nullable. However, the implied default nullability of a type of a site may be overridden by the following provision.

If a Syntax Rule *RULE* originally determines (infers) the declared type of a site *S* to be some type *T* without explicitly specifying the nullability of *T*, then

Case:

- 1) If an implementation-defined (IW022) mechanism immediately after the determination made by *RULE* regarding the nullability of the declared type of *S* establishes that the null value is not going to be assigned to *S* by the application of General Rules, then the declared type of *S* is specialized to be the material variant of *T* (instead of *T*).

NOTE 159 — This effectively overrides the original determination made by *RULE* regarding the nullability of the declared type of *S*.

- 2) Otherwise, the declared type of *S* (implicitly) is the nullable variant of *T*.

NOTE 160 — This preserves the original determination made by *RULE* regarding the nullability of the declared type of *S*.

NOTE 161 — These provisions only allow the relaxation of Syntax Rules that determine (infer) declared types. Syntax Rules imposing requirements on declared types of sites are not affected.

5 Notation and conventions

5.1 Notation taken from The Unicode® Standard

The notation for the representation of UCS code points and sequences of code points is defined in [The Unicode® Standard](#), Appendix A, “Notational Conventions”.

In this document, this notation is used only to unambiguously identify characters and is not meant to imply a specific encoding for any GQL-implementation’s use of that character.

5.2 Notation

The syntactic notation used in this document is an extended version of BNF (“Backus Normal Form” or “Backus Naur Form”).

In a BNF language definition, each syntactic element, known as a *BNF non-terminal symbol*, of the language is defined by means of a *production rule*. This defines the syntactic element in terms of a formula consisting of the characters, character strings, and elements that can be used to form an instance, known as a *BNF non-terminal instance*, of it.

In the version of BNF used in this document, the following symbols have the meanings shown in [Table 2](#), “Symbols used in BNF”.

Table 2 — Symbols used in BNF

Symbol	Meaning
< >	A character string enclosed in angle brackets is the name of a syntactic element (i.e., the name of a BNF non-terminal symbol) of the GQL language.
::=	The definition operator is used in a production rule to separate the element defined by the rule from its definition. The element being defined appears to the left of the operator and the formula that defines the element appears to the right.
[]	Square brackets indicate optional elements in a formula. The portion of the formula within the brackets may be explicitly specified or may be omitted.
{ }	Braces group elements in a formula. The portion of the formula within the braces shall be explicitly specified.
	The alternative operator. The vertical bar indicates that the portion of the formula following the bar is an alternative to the portion preceding the bar. If the vertical bar appears at a position where it is not enclosed in braces or square brackets, it specifies a complete alternative for the element defined by the production rule. If the vertical bar appears in a portion of a formula enclosed in braces or square brackets, it specifies alternatives for the content of the innermost pair of such braces or brackets.

ISO/IEC 39075:2024(en)
5.2 Notation

Symbol	Meaning
...	The ellipsis indicates that the element to which it applies in a formula may be repeated any number of times. If the ellipsis appears immediately after a closing brace “}”, then it applies to the portion of the formula enclosed between that closing brace and the corresponding opening brace “{”. If an ellipsis appears after any other element, then it applies only to that element. In Syntax Rules, General Rules, and Conformance Rules, a reference to the <i>n</i> -th element in such a list assumes the order in which these are specified, unless otherwise stated.
!!	Introduces either a reference to the Syntax Rules, used when the definition of a syntactic element is not expressed in BNF, or the Unicode code point or code point sequence that define the character(s) of the BNF production.

Whitespace is used to separate syntactic elements. Multiple whitespace characters are treated as a single space. Apart from those symbols to which special functions were given above, other characters and character strings in a formula stand for themselves. In addition, if the symbols to the right of the definition operator in a production consist entirely of BNF symbols, then those symbols stand for themselves and do not take on their special meaning.

Pairs of braces and square brackets may be nested to any depth, and the alternative operator may appear at any depth within such a nest.

A character string that forms an instance of a syntactic element may be generated from the BNF definition of that element by application of the following steps:

- 1) Select any one option from those defined in the right hand side of a production rule for the element, and replace the element with this option.
- 2) Replace each ellipsis and the object to which it applies with one or more instances of that object.
- 3) For every portion of the character string enclosed in square brackets, either delete the brackets and their contents or change the brackets to braces.
- 4) For every portion of the character string enclosed in braces, apply [Step 1\)](#) through [Step 5\)](#) to the substring between the braces, then remove the braces.
- 5) Apply [Step 1\)](#) through [Step 5\)](#) to any element whose BNF definition generates what remains in the character string.

The expansion or production is complete when there is no further syntactic element whose BNF definition generates what remains in the character string.

The left normal form derivation of a character string *CS* in the GQL source text character repertoire that is generated from the BNF definition of a BNF non-terminal symbol *NT* is obtained by applying [Step 1\)](#) through [Step 5\)](#) above to *NT*, always selecting in [Step 5\)](#) the leftmost BNF non-terminal symbol.

The syntax defined in this document is available from the ISO website as a “digital artifact”. See <https://standards.iso.org/iso-iec/39075/ed-1/en/> to download digital artifacts for this document. To download the syntax defined in a plain-text format, select the file named [ISO_IEC_39075\(en\).bnf.txt](#). To download the syntax defined in an XML format, select the file named [ISO_IEC_39075\(en\).bnf.xml](#).

5.3 Conventions

5.3.1 Specification of syntactic elements

Syntactic elements are specified in terms of:

- **Function:** A short statement of the purpose of the element.
- **Format:** A BNF definition of the syntax of the element.
- **Syntax Rules:** A specification in English of the syntactic properties of the element. These include rules for the following aspects that are specified in the sequence given:
 - The expansion of syntactic short-hand forms.
 - The normalization of syntactic forms.
 - Additional syntactic constraints, not expressed in BNF, that the element shall satisfy.
 - The visibility or scope of identifiers.
 - Specifying or defining the type of elements.
- **General Rules:** A specification in English of the run-time effect of the element. Where more than one General Rule is used to specify the effect of an element, the required effect is that which would be obtained by beginning with the first General Rule and applying the Rules in numeric sequence unless a Rule is applied that specifies or implies a change in sequence or termination of the application of the Rules. Unless otherwise specified or implied by a specific Rule that is applied, application of General Rules terminates when the last in the sequence has been applied.
- **Conformance Rules:** A specification of how the element shall be supported for conformance to GQL. Conformance Rules are effectively a kind of Syntax Rule, differentiated only because of their use to specify conformance to the GQL language. Conformance Rules in a given Subclause are therefore always applied before the Syntax Rules of that Subclause, and are not normally applied to the result of syntactic transformations. However, in a few cases, Conformance Rules are explicitly applied to syntactic transformations as defined in the Syntax Rules.

The scope of notational symbols is the Subclause in which those symbols are defined. Within a Subclause, the symbols defined in Syntax Rules or General Rules can be referenced in other rules provided that they are defined before being referenced.

5.3.2 Use of terms

5.3.2.1 Syntactic containment

Let $\langle A \rangle$, $\langle B \rangle$, and $\langle C \rangle$ be syntactic elements; let $A1$, $B1$, and $C1$ respectively be instances of $\langle A \rangle$, $\langle B \rangle$, and $\langle C \rangle$.

In a Format, $\langle A \rangle$ is said to *immediately contain* $\langle B \rangle$ if $\langle B \rangle$ appears on the right-hand side of the BNF production rule for $\langle A \rangle$. An $\langle A \rangle$ is said to *contain* or *specify* $\langle C \rangle$ if $\langle A \rangle$ immediately contains $\langle C \rangle$ or if $\langle A \rangle$ immediately contains a $\langle B \rangle$ that contains $\langle C \rangle$.

In GQL language, $A1$ is said to *immediately contain* $B1$ if $\langle A \rangle$ immediately contains $\langle B \rangle$ and $B1$ is part of the text of $A1$. $A1$ is said to *contain* or *specify* $C1$ if $A1$ immediately contains $C1$ or if $A1$ immediately contains $B1$ and $B1$ contains $C1$. If $A1$ contains $C1$, then $C1$ is *contained in* $A1$ and $C1$ is *specified by* $A1$.

$A1$ is said to contain $B1$ *with an intervening instance of* $\langle C \rangle$ if $A1$ contains $B1$ and $A1$ contains an instance of $\langle C \rangle$ that contains $B1$. $A1$ is said to contain $B1$ *without an intervening instance of* $\langle C \rangle$ if $A1$ contains $B1$ and $A1$ does not contain an instance of $\langle C \rangle$ that contains $B1$.

$A1$ *simply contains* $B1$ if $A1$ contains $B1$ without an intervening instance of $\langle A \rangle$ or an intervening instance of $\langle B \rangle$. If $A1$ simply contains $B1$, then $B1$ is *simply contained in* $A1$.

$A1$ *directly contains* $B1$ if $A1$ contains $B1$ without an intervening instance of $\langle \text{procedure body} \rangle$. If $A1$ directly contains $B1$, then $B1$ is *directly contained in* $A1$.

ISO/IEC 39075:2024(en)
5.3 Conventions

If an instance of $\langle A \rangle$ contains an instance of $\langle B \rangle$, then $\langle B \rangle$ is said to be *contained in* $\langle A \rangle$ and $\langle A \rangle$ is said to be a *containing* production symbol for $\langle B \rangle$. If an instance of $\langle A \rangle$ simply contains an instance of $\langle B \rangle$, then $\langle B \rangle$ is said to be *simply contained in* $\langle A \rangle$ and $\langle A \rangle$ is said to be a *simply containing* production symbol for $\langle B \rangle$. If an instance of $\langle A \rangle$ directly contains an instance of $\langle B \rangle$, then $\langle B \rangle$ is said to be *directly contained in* $\langle A \rangle$ and $\langle A \rangle$ is said to be a *directly containing* production symbol for $\langle B \rangle$.

$A1$ is the *innermost* $\langle A \rangle$ satisfying a condition C if $A1$ satisfies C and $A1$ does not contain an instance of $\langle A \rangle$ that satisfies C . $A1$ is the *outermost* $\langle A \rangle$ satisfying a condition C if $A1$ satisfies C and $A1$ is not contained in an instance of $\langle A \rangle$ that satisfies C .

In a Format, the verb “to be” (including all its grammatical variants, such as “is”) is defined as follows. $\langle A \rangle$ is said to be $\langle B \rangle$ if there exists a BNF production rule of the form $\langle A \rangle ::= \langle B \rangle$. If $\langle A \rangle$ is $\langle B \rangle$ and $\langle B \rangle$ is $\langle C \rangle$, then $\langle A \rangle$ is $\langle C \rangle$. If $\langle A \rangle$ is $\langle C \rangle$, then $\langle C \rangle$ is said to *constitute* $\langle A \rangle$. In GQL language, $A1$ is said to be $B1$ if $\langle A \rangle$ is $\langle B \rangle$ and the text of $A1$ is the text of $B1$. Conversely, $B1$ is said to *constitute* $A1$ if $A1$ is $B1$.

5.3.2.2 Keywords and $\langle \text{keyword} \rangle$ s

In the text (including the Format sections) of this document, certain syntax elements are used to specify GQL statements and parts of GQL expressions. Those syntax elements are called “keyword”s. They are used either to identify the GQL statement being specified or to identify details of the syntax of statements and expressions.

In this document, the word “keyword” is used to refer to those syntax elements, and the BNF non-terminal symbol “ $\langle \text{keyword} \rangle$ ” is used when the definition or use of a keyword in a Format is referenced. Throughout this document, all keywords are specified using only upper-case letters; for example, “MATCH” is a keyword.

In GQL syntax, a keyword spelled using one or more lower-case letters is equivalent to spelling it with every lower-case letter replaced with its corresponding upper-case letter. For example, “mATch”, when used as a keyword, is identical in effect to “MATCH”.

5.3.2.3 Terms denoting rule requirements

In the Syntax Rules, the term *shall* defines conditions that are required to be true of syntactically conforming GQL language. When such conditions depend on the contents or the structure of the GQL-catalog or the descriptors of GQL-objects reachable via the GQL-catalog, the GQL-session context, or the GQL-request context, they are required to be true just before the actions specified by the General Rules are performed. The treatment of language that does not conform to the Formats and Syntax Rules is implementation-defined (IE005). If any condition required by Syntax Rules is not satisfied when the application of General Rules is attempted and the GQL-implementation is neither processing non-conforming GQL language nor processing conforming GQL language in a non-conforming manner, then an exception condition is raised as specified in Subclause 4.7.3, “Execution of GQL-requests”.

In the Conformance Rules, the term *shall* defines conditions that are required to be satisfied if the named Feature is or Features are not supported.

5.3.2.4 Rule evaluation order

A conforming GQL-implementation is not required to perform the exact sequence of actions defined in the General Rules, provided its effect on GQL-data and the GQL-catalog is identical to the effect of that sequence. The term *effectively* is used to emphasize actions whose effect may be achieved in other ways by a GQL-implementation.

The Conformance Rules for contained syntactic elements are effectively applied at the same time as the Conformance Rules for the containing syntactic elements. Similarly the Syntax Rules for contained syntactic elements are effectively applied at the same time as the Syntax Rules for the containing syntactic elements.

ISO/IEC 39075:2024(en)

5.3 Conventions

The General Rules for contained syntactic elements are effectively applied before the General Rules for the containing syntactic elements.

Where the precedence of operators is determined by the Formats of this document or by parentheses, those operators are effectively applied in the order specified by that precedence.

Where the precedence is not determined by the Formats or by parentheses, effective evaluation of expressions is *generally* performed from left to right. However, it is implementation-dependent (US008) whether expressions are *actually* evaluated left to right, particularly when the evaluation of operands or operators causes conditions to be raised or if the results of the expressions can be determined without completely evaluating all parts of the expression.

If some syntactic element contains more than one other syntactic element, then the General Rules for contained elements that appear earlier in the production for the containing syntactic element are applied before the General Rules for contained elements that appear later.

For example, in the production:

```
<A> ::=
  <B> <C>

<B> ::=
  ...

<C> ::=
  ...
```

the Conformance Rules for <A>, , and <C> are effectively applied simultaneously. Similarly the Syntax Rules for <A>, , and <C> are effectively applied simultaneously. The General Rules for are applied before the General Rules for <C>, and the General Rules for <A> are applied after the General Rules for both and <C>.

An exception to this rule is when the General Rules of the containing syntactic element explicitly states when the General Rules of the contained syntactic element are to be applied.

NOTE 162 — In this document these exceptions are shown by the presence of a General Rule of the form “The General Rules of xxx are applied.” where xxx is a BNF non-terminal symbol or denotes a BNF non-terminal instance of a BNF non-terminal symbol. This indicates that xxx is evaluated at this point and not at the point implied by the general “Rule evaluation order”.

If the result of an expression or search condition is not dependent on the result of some part of that expression or search condition, then that part of the expression or search condition is said to be *inessential*.

If evaluation of an inessential part would cause an exception condition to be raised, then it is implementation-dependent (UA004) whether that exception condition is raised.

During the computation of the result of an expression, the GQL-implementation may produce one or more *intermediate results* that are used in determining that result. The declared type of a site that contains an intermediate result is implementation-dependent (UV007).

5.3.2.5 Conditional rules

A conditional rule is specified with “If” or “Case” conventions. A rule specified with “Case” conventions includes a list of conditional subrules using “If” conventions. The first such “If” subrule whose condition is true is the effective subrule of the “Case” rule. The last subrule of a “Case” rule may specify “Otherwise”, in which case it is the effective subrule of the “Case” rule if no preceding “If” subrule in the “Case” rule is satisfied. If the last subrule does not specify “Otherwise”, and if there is no subrule whose condition is true, then there is no effective subrule of the “Case” rule.

5.3.2.6 Syntactic substitution

In the Syntax and General Rules, the phrase “*X is implicit*” indicates that the Syntax and General Rules are to be interpreted as if the element *X* had actually been specified. Within the Syntax Rules of a given Subclause, it is known whether the element was explicitly specified or is implicit.

In the Syntax and General Rules, the phrase “the following <A> is implicit: *Y*” indicates that the Syntax and General Rules are to be interpreted as if a syntactic element <A> containing *Y* had actually been specified.

In the Syntax Rules and General Rules, the phrase “*former is equivalent to latter*” indicates that the Syntax Rules and General Rules are to be interpreted as if all instances of *former* in the element had been instances of *latter*.

If a BNF non-terminal symbol is referenced in a Subclause (e.g., to identify a BNF non-terminal instance by specifying the BNF non-terminal symbol) without specifying how it is contained in a BNF production that the Subclause defines, then

Case:

- If the BNF non-terminal symbol is itself defined in the Subclause, then the reference shall be assumed to be to the occurrence of that BNF non-terminal symbol on the left side of the defining production.
- Otherwise, the reference shall be assumed to be to a BNF production in which the particular BNF non-terminal symbol is immediately contained.

5.3.3 Descriptors

A descriptor is a coded description of the metadata of a GQL-object. The concept of descriptor is used in specifying the semantics of GQL. It is not necessary that any descriptor exist in any particular form in any GQL-environment.

Some GQL-objects cannot exist except in the context of other GQL-objects. For example, nodes cannot exist except within the context of graphs. Each such object is independently described by its own descriptor, and the descriptor of an enabling object (e.g., graph) is said to *include* the descriptor of each enabled object (e.g., node or edge). Conversely, the descriptor of an enabled object is said to *be included in* the descriptor of an enabling object.

In other cases, certain GQL-objects cannot exist unless some other GQL-object exists, even though there is no inclusion relationship. In general, a descriptor *D1* can be said to depend on, or to be dependent on, some descriptor *D2*.

Some GQL-objects are also dictionaries that uniquely associate an identifier to certain enabled objects. Such named subobjects are said to be held by such a dictionary. The descriptor of a GQL-object that is also a dictionary represents such identifier associations as a set of *named subobjects* whose elements are (identifier, object descriptor) pairs. The object descriptor in such a pair is considered included in the descriptor of the GQL-object that holds it.

The execution of a statement can result in the creation of many descriptors. A GQL-object that is created as a result of a statement can depend on other descriptors that are only created as a result of the execution of that statement.

There are two ways of indicating dependency of one GQL-object on another. In many cases, the descriptor of the dependent GQL-object is said to “include the name of” the GQL-object on which it is dependent. In this case “the name of” is to be understood as meaning “sufficient information to identify the descriptor of”. Alternatively, the descriptor of the dependent GQL-object can be said to include text of the GQL-object on which it is dependent. However, in such cases, whether the GQL-implementation includes actual text (with defaults and implications made explicit) or its own style of parse tree is irrelevant; the validity of the descriptor is clearly “dependent on” the existence of descriptors of objects that are referenced in it.

An attempt to destroy a GQL-object, and hence its descriptor, can fail if other descriptors are dependent on it, depending on how the destruction is specified. Such an attempt can also fail if the descriptor to be destroyed is included in some other descriptor. Destruction of a descriptor results in the destruction of all descriptors included in it, but has no effect on descriptors on which it is dependent.

The implementation of some GQL-objects described by descriptors requires the existence of objects not specified in this document. Where such objects are required, they are effectively created whenever the associated descriptor is created and effectively destroyed whenever the associated descriptor is destroyed.

5.3.4 Subclauses used as subroutines

In this document, some Subclauses are defined without explicit syntax to invoke their semantics. Such Subclauses, called subroutine Subclauses, typically factor out rules that are required by one or more other Subclauses and are intended to be invoked by the rules of those other Subclauses.

In other words, the rules of these Subclauses behave as though they were a sort of definitional “subroutine” that is invoked by other Subclauses. These subroutine Subclauses are typically specified in a manner that requires information to be passed to them from their invokers. The information that is required to be passed is represented as parameters of these subroutine Subclauses, and that information is required to be passed in the form of arguments provided by the invokers of these subroutine Subclauses.

Every invocation of a subroutine Subclause shall explicitly provide information for every required parameter of the subroutine Subclause being invoked.

In Subclauses that have Subclause Signatures, those signatures are followed by *non-normative* text that documents every parameter and every returned value contained in that signature. That text is intended solely to assist readers of the document in their understanding of the Subclause Signatures and their parameters and returns. If the descriptive text and the normative text in those Subclauses disagree, the Subclauses’ normative text is authoritative.

NOTE 163 — In this document the invocation will occur in the form “The xxx Rules of Subclause *n.n*, “aaaaaaa”, with ...”.

5.3.5 Document typography

In the text of this document, the following typographic conventions are used:

- *Italicized text* is used for several purposes:
 - representations of GQL truth values (e.g., *True*);
 - representations of symbolic variables, both their definitions and their uses;
 - textual definitions of important terms and concepts.
- **Bold text** is used to display of terms taken from other standards that display those terms in bold type.
- Underlined text is used in the representation of GQL truth values (e.g., *True*).

5.3.6 Index typography

In the Indexes to this document, the following conventions are used:

- An index entry in **boldface** indicates the page where the word, phrase, or BNF non-terminal symbol is defined.
- An index entry in *italics* indicates a page where the BNF non-terminal symbol is used in a Format.
- An index entry in neither boldface nor italics indicates a page where the word, phrase, or BNF non-terminal symbol is not defined, but is used other than in a Format (e.g., in a heading, Function,

Syntax Rule, General Rule, Conformance Rule, Table, or other descriptive text). For this purpose, text that identifies a BNF non-terminal instance by specifying a BNF non-terminal symbol is considered a use of that BNF non-terminal symbol.

5.3.7 Feature ID and Feature Name

Features are either *standard-defined features* or implementation-defined features.

Standard-defined features are defined in this document. Implementation-defined features are defined by GQL-implementations (see [Subclause 24.5.3, “Extensions and options”](#)).

Features are referenced by a Feature ID and by a Feature Name. A Feature ID value comprises a letter and three digits.

Feature IDs whose letter is “V” are reserved for implementation-defined features.

Standard-defined features are optional features that are defined by explicit or implicit Conformance Rules. An implicit Conformance Rule is any normative statement that begins “Without Feature *nnn*,”. The Feature ID of a standard-defined feature is stable and can be depended on to remain constant.

For convenience, all of the features defined in this document are collected together in a non-normative annex, [Annex D, “GQL optional feature taxonomy”](#).

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

6 <GQL-program>

Function

Specify a conforming GQL-program.

Format

```
<GQL-program> ::=
    <program activity> [ <session close command> ]
    | <session close command>

<program activity> ::=
    <session activity>
    | <transaction activity>

<session activity> ::=
    <session reset command>...
    | <session set command>... [ <session reset command>... ]

<transaction activity> ::=
    <start transaction command>
    [ <procedure specification> [ <end transaction command> ] ]
    | <procedure specification> [ <end transaction command> ]
    | <end transaction command>

<end transaction command> ::=
    <rollback command>
    | <commit command>
```

Syntax Rules

NOTE 164 — See Subclause 4.7.3, “Execution of GQL-requests”, regarding operations performed prior to the application of these Syntax Rules.

- 1) Let *PROG* be the <GQL-program>.
- 2) Let *IS* and *IG* be the initial session and the initial graph, respectively, of *PROG*.

NOTE 165 — The initial session and the initial graph of a <GQL-program> are determined by the Syntax Rules of Subclause 22.1, “Annotation of a <GQL-program>”.

- 3) The scope clause of *PROG* is *PROG*.
- 4) For every pair of instances *I1* and *I2* of a <non-delimited identifier> or a <delimited identifier> contained in *PROG*, if *I1* and *I2* are visually confusable with each other, then an exception condition is raised: *syntax error or access rule violation — use of visually confusable identifiers (42004)*.
- 5) Let *ACSR* be a (possibly “omitted”) <absolute catalog schema reference> defined as follows.

Case:

- a) If *IS* is not “omitted” and identifies a GQL-schema, then *ACSR* is *IS*.
- b) Otherwise, *ACSR* is “omitted”.

- 6) If *ACSR* is not “omitted”, then:
 - a) The scope of *ACSR* comprises *PROG*.
 - b) *PROG* identifies *ACSR* as a working schema reference.
- 7) Let *WGS* be defined as follows.
Case:
 - a) If the current session graph is “not set”, then:
 - i) *WGS* is a site occupied by the graph reference value *IG*.
 - ii) If the referent of *IG* is a graph that has a constraining graph type *GT*, then the declared type of *WGS* is the graph reference value type whose constraining object type is *GT*; otherwise, the declared type of *WGS* is the open graph reference value type.
 - b) Otherwise, *WGS* is “omitted” and *WGS* is a site occupied by the graph reference value *IG*.
- 8) If *WGS* is not “omitted”, then:
 - a) The scope of *WGS* comprises *PROG*.
 - b) *PROG* identifies *WGS* as a working graph site.
- 9) The declared type of incoming working record of *PROG* is the material unit record type.
- 10) The declared type of incoming working table of *PROG* is the material unit binding table type.

General Rules

NOTE 166 — See Subclause 4.7.3, “Execution of GQL-requests”, regarding operations performed prior to the application of these General Rules.

- 1) If *PROG* simply contains the <program activity> *PACT*, then the General Rules of *PACT* are applied.
- 2) If *PROG* simply contains the <session activity> *SACT*, then:
Case:
 - a) If the current transaction is “not set”, then:
 - i) The current transaction is set to a newly initiated GQL-transaction.
NOTE 167 — This determines the currently active GQL-transaction associated with the GQL-session of the currently executing GQL-request.
 - ii) The General Rules of *SACT* are applied.
 - iii) The currently active GQL-transaction is terminated and the current transaction is set to “not set”.
 - b) Otherwise, the General Rules of *SACT* are applied.
- 3) If *PROG* simply contains the <session close command> *SCC*, then the General Rules of *SCC* are applied.

Conformance Rules

- 1) Without Feature GT01, “Explicit transaction commands”, conforming GQL language shall not contain a <start transaction command> or an <end transaction command>.

7 Session management

7.1 <session set command>

Function

Set values in the session context.

Format

```

<session set command> ::=
  SESSION SET {
    <session set schema clause>
  | <session set graph clause>
  | <session set time zone clause>
  | <session set parameter clause>
  }

<session set schema clause> ::=
  SCHEMA <schema reference>

<session set graph clause> ::=
  [ PROPERTY ] GRAPH <graph expression>

<session set time zone clause> ::=
  TIME ZONE <set time zone value>

<set time zone value> ::=
  <time zone string>

<session set parameter clause> ::=
  <session set graph parameter clause>
  | <session set binding table parameter clause>
  | <session set value parameter clause>

<session set graph parameter clause> ::=
  [ PROPERTY ] GRAPH <session set parameter name> <opt typed graph initializer>

<session set binding table parameter clause> ::=
  [ BINDING ] TABLE <session set parameter name> <opt typed binding table initializer>

<session set value parameter clause> ::=
  VALUE <session set parameter name> <opt typed value initializer>

<session set parameter name> ::=
  [ IF NOT EXISTS ] <session parameter specification>

```

Syntax Rules

- 1) Let *SSC* be the <session set command>.
- 2) Let *PROG* be the <GQL-program> that simply contains *SSC*.
- 3) The declared type of *SSC* is the empty type.

ISO/IEC 39075:2024(en)
7.1 <session set command>

- 4) If *SSC* simply contains the <session set schema clause> *SSSC*, then let *SR* be the <schema reference> immediately contained in *SSSC*.
- 5) If the <set time zone value> *STZV* is specified, then let *TZD* be the time zone displacement specified by the <time zone string> immediately contained in *STZV*.
- 6) If *SSC* simply contains the <session set parameter clause>, then:
 - a) Let *SSPN* be the <session set parameter name> simply contained in *SSC*, let *SPS* be the <session parameter specification> immediately contained in *SSPN*, and let *PN* be the parameter name of *SPS*.
 - b) If *SSPN* contains IF NOT EXISTS and *SPS* references a defined session parameter, then no further Syntax Rules of this Subclause are applied.
 - c) Let the value type *PVT* be defined as follows.

Case:

- i) If *SSC* is the <session set graph parameter clause> *SSGPC*, then:

Case:

- 1) Let *OTGI* be the <opt typed graph initializer> immediately contained in *SSGPC* and let *GI* be the <graph initializer> immediately contained in *OTGI*.
- 2) Let *GIT* be the declared type of *GI*.
- 3) Case:
 - A) If *OTGI* immediately contains a <graph reference value type> *OGT*, then:
 - I) *GIT* shall be assignable to the type specified by *OGT*.
 - II) *PVT* is the type specified by *OGT*.
 - B) Otherwise, *PVT* is *GIT*.

- ii) If *SSC* is the <session set binding table parameter clause> *SSBTPC*, then:

Case:

- 1) Let *OTBTI* be the <opt typed binding table initializer> immediately contained in *SSBTPC* and let *BTI* be the <binding table initializer> immediately contained in *OTBTI*.
- 2) Let *BTIT* be the declared type of *BTI*.
- 3) Case:
 - A) If *OTBTI* immediately contains a <binding table reference value type> *OBTT*, then:
 - I) *BTIT* shall be assignable to the type specified by *OBTT*.
 - II) *PVT* is the type specified by *OBTT*.
 - B) Otherwise, *PVT* is *BTIT*.

- iii) If *SSC* is the <session set value parameter clause> *SSVPC*, then:

Case:

ISO/IEC 39075:2024(en)
7.1 <session set command>

- 1) Let *OTVI* be the <opt typed value initializer> immediately contained in *SSVPC* and let *VI* be the <value initializer> immediately contained in *OTVI*.
- 2) Let *VIT* be the declared type of *VI*.
- 3) Case:
 - A) If *OTVI* immediately contains a <value type> *OVT*, then:
 - I) *VIT* shall be assignable to the type specified by *OVT*.
 - II) *PVT* is the type specified by *OVT*.
 - B) Otherwise, *PVT* is *VIT*.
- d) Case:
 - i) If *SPS* references a defined session parameter, then *PVT* shall be assignable to the parameter value type of *SPS*.
 - ii) Otherwise, *SPS* does not reference a defined session parameter.Case:
 - 1) If there are other <session set command>s simply contained in *PROG* that precede *SSC* in *PROG* and that simply contain a <session parameter specification> whose parameter name is *PN*, then:
 - A) Let *DT* be the declared type of the <session parameter specification> simply contained in the rightmost such <session set command>.
 - B) *PVT* shall be assignable to *DT*.
 - C) The declared type of *SPS* is *DT*.
 - 2) Otherwise, *SSC* is the initial (leftmost) <session set command> simply contained in *PROG* for *SPS* and the declared type of *SPS* is *PVT*.

General Rules

- 1) If *SSC* simply contains the <session set schema clause>, then the current session schema is set to an <absolute catalog schema reference> that identifies the GQL-schema identified by *SR*.
- 2) If *SSC* simply contains the <session set graph clause> *SSGC*, then the current session graph is set to the graph reference value that is the result of the <graph expression> immediately contained in *SSGC*.
- 3) If *SSC* simply contains the <session set time zone clause>, then the current session time zone displacement is set to *TZD*.
- 4) If *SSC* simply contains the <session set parameter clause>, then

Case:

- a) If *SSPN* contains IF NOT EXISTS and the current session context contains a session parameter with parameter name *PN*, then the current application of General Rules of this Subclause resumes with [General Rule 5](#)).
- b) Let *PV* be defined as follows.

Case:

- i) If *SSC* is the <session set graph parameter clause> *SSGPC*, then *PV* is the result of *GI*.
 - ii) If *SSC* is the <session set binding table parameter clause> *SSBTPC*, then *PV* is the result of *BTI*.
 - iii) Otherwise, *SSC* is the <session set value parameter clause> *SSVPC* and *PV* is the result of *VI*.
- c) Case:
- i) If the current session context contains a session parameter *SP* with parameter name *PN*, then *PV* is assigned to the parameter value of *SP*.
 - ii) Otherwise, a new session parameter with parameter name *PN*, parameter value *PV*, and parameter value type *PVT* is created in the current session context.
- 5) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GS01, "SESSION SET command: session-local graph parameters", conforming GQL language shall not contain a <session set graph parameter clause>.
- 2) Without Feature GS02, "SESSION SET command: session-local binding table parameters", conforming GQL language shall not contain a <session set binding table parameter clause>.
- 3) Without Feature GS03, "SESSION SET command: session-local value parameters", conforming GQL language shall not contain a <session set value parameter clause>.
- 4) Without Feature GS10, "SESSION SET command: session-local binding table parameters based on subqueries", conforming GQL language shall not contain a <session set binding table parameter clause> that contains a <procedure body>.
- 5) Without Feature GS11, "SESSION SET command: session-local value parameters based on subqueries", conforming GQL language shall not contain a <session set value parameter clause> that contains a <procedure body>.
- 6) Without Feature GS12, "SESSION SET command: session-local graph parameters based on simple expressions or references", conforming GQL language shall not contain a <session set graph parameter clause> that simply contains a <graph expression> that does not conform to <value specification> or is a <graph reference>.
- 7) Without Feature GS13, "SESSION SET command: session-local binding table parameters based on simple expressions or references", conforming GQL language shall not contain a <session set binding table parameter clause> that simply contains a <binding table expression> that does not conform to <value specification> or is a <binding table reference>.
- 8) Without Feature GS14, "SESSION SET command: session-local value parameters based on simple expressions", conforming GQL language shall not contain a <session set value parameter clause> that simply contains a <value expression> that does not conform to <value specification>.
- 9) Without Feature GS15, "SESSION SET command: set time zone displacement", conforming GQL language shall not contain a <session set time zone clause>.

7.2 <session reset command>

Function

Reset session parameters and characteristics.

Format

```
<session reset command> ::=  
    SESSION RESET [ <session reset arguments> ]  
  
<session reset arguments> ::=  
    [ ALL ] { PARAMETERS | CHARACTERISTICS }  
    | SCHEMA  
    | [ PROPERTY ] GRAPH  
    | TIME_ZONE  
    | [ PARAMETER ] <session parameter specification>
```

Syntax Rules

- 1) Let *SRC* be the <session reset command>.
- 2) Case:
 - a) If *SRC* simply contains <session reset arguments> *SRA*, then
Case:
 - i) If *SRA* is CHARACTERISTICS or PARAMETERS, then *SRA* is effectively replaced by:
ALL *SRA*
 - ii) If *SRA* is GRAPH, then *SRA* is effectively replaced by:
PROPERTY GRAPH
 - b) Otherwise, *SRC* is effectively replaced by:
SESSION RESET ALL CHARACTERISTICS
- 3) If *SRC* simply contains a <session parameter specification> *SPC*, then the declared type of *SPC* is the empty type.
- 4) The declared type of *SRC* is the empty type.

General Rules

- 1) If ALL CHARACTERISTICS or SCHEMA is specified and the current home schema is not “omitted”, then the current session schema is set to the current home schema; otherwise, the current session schema is set to “not set”.
- 2) If ALL CHARACTERISTICS or PROPERTY GRAPH is specified and the current home graph is not “omitted”, then the current session graph is set to the current home graph; otherwise, the current session graph is set to “not set”.
- 3) If ALL CHARACTERISTICS or TIME_ZONE is specified, then the current session time zone displacement is set to the implementation-defined (ID048) default time zone displacement.

- 4) If ALL CHARACTERISTICS or ALL PARAMETERS is specified, then:
 - a) Each session parameter is removed from the current session context.
 - b) Each implementation-defined (ID049) default session parameter is added to the current session context.
- 5) If the <session parameter specification> SPS is specified, then:
 - a) Let *PN* be the parameter name of *SPS*.
 - b) If the current session context has a session parameter *SP* whose parameter name is *PN*, then *SP* is removed from the current session context.
 - c) If *PN* specifies the parameter name of an implementation-defined (ID049) default session parameter *DP*, then *DP* is added to the current session context.
- 6) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GS04, "SESSION RESET command: reset all characteristics", conforming GQL language shall contain a <session reset arguments>.
- 2) Without Feature GS04, "SESSION RESET command: reset all characteristics", conforming GQL language shall not contain a <session reset arguments> that contains either PARAMETERS or CHARACTERISTICS.
- 3) Without Feature GS05, "SESSION RESET command: reset session schema", conforming GQL language shall not contain SESSION RESET SCHEMA.
- 4) Without Feature GS06, "SESSION RESET command: reset session graph", conforming GQL language shall not contain SESSION RESET PROPERTY GRAPH or SESSION RESET GRAPH.
- 5) Without Feature GS07, "SESSION RESET command: reset time zone displacement", conforming GQL language shall not contain SESSION RESET TIME ZONE.
- 6) Without Feature GS08, "SESSION RESET command: reset all session parameters", conforming GQL language shall not contain SESSION RESET ALL PARAMETERS or SESSION RESET PARAMETER.
- 7) Without Feature GS16, "SESSION RESET command: reset individual session parameters", conforming GQL language shall not contain a <session reset arguments> that immediately contains a <parameter name>.

7.3 <session close command>

Function

Close the current session.

Format

```
<session close command> ::=  
    SESSION CLOSE
```

Syntax Rules

- 1) The declared type of the <session close command> is the empty type.

General Rules

- 1) The current termination flag is set to *True*.
- 2) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

None.

7.4 <session parameter specification>

Function

Specify a session parameter for a value.

Format

<session parameter specification> ::=
 <general parameter reference>

Syntax Rules

NOTE 168 — The Syntax Rules of Subclause 22.1, “Annotation of a <GQL-program>” determine whether a <session parameter specification> references a defined session parameter and, if that is the case, its parameter name and parameter value type.

NOTE 169 — The declared type of a <session parameter specification> is determined by the Syntax Rules of Subclause 7.1, “<session set command>”, and Subclause 7.2, “<session reset command>”.

None.

General Rules

None.

Conformance Rules

None.

8 Transaction management

8.1 <start transaction command>

Function

Start a new GQL-transaction and set its characteristics.

Format

```
<start transaction command> ::=  
  START TRANSACTION [ <transaction characteristics> ]
```

Syntax Rules

- 1) Let *TS* be the <start transaction command>.
- 2) If *TS* does not simply contain <transaction characteristics>, then an implementation-defined (ID006) default transaction characteristics that simply contains a <transaction mode> READ WRITE is implicit.

General Rules

- 1) If a GQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active GQL-transaction (25601)*.
- 2) Let *TC* be the explicit or implicit <transaction characteristics> simply contained in *TS*.
- 3) A new GQL-transaction *TX* with the transaction characteristics specified by *TC* as its transaction characteristics is initiated.

NOTE 170 — Every transaction records its transaction characteristics.

- 4) The current transaction is set to *TX*.

NOTE 171 — This determines *TX* as the currently active GQL-transaction associated with the GQL-session of the currently executing GQL-request.

Conformance Rules

- 1) Without Feature GT02, “Specified transaction characteristics”, conforming GQL language shall not contain a <start transaction command> that contains <transaction characteristics>.

8.2 <transaction characteristics>

Function

Specify GQL-transaction characteristics.

Format

```
<transaction characteristics> ::=  
  <transaction mode> [ { <comma> <transaction mode> }... ]  
  
<transaction mode> ::=  
  <transaction access mode>  
  | <implementation-defined access mode>  
  
<transaction access mode> ::=  
  READ ONLY  
  | READ WRITE  
  
<implementation-defined access mode> ::=  
  !! See the Syntax Rules.
```

Syntax Rules

- 1) Let *TC* be the <transaction characteristics>.
- 2) *TC* shall contain exactly one <transaction access mode>.
- 3) The Format and Syntax Rules for <implementation-defined access mode> are implementation-defined (IE002).

General Rules

None.

Conformance Rules

None.

8.3 <rollback command>

Function

Terminate the currently active GQL-transaction with rollback.

Format

<rollback command> ::=
ROLLBACK

Syntax Rules

None.

General Rules

- 1) If the currently active GQL-transaction is part of an encompassing transaction that is controlled by an agent other than the GQL-agent and the <rollback command> is not being implicitly executed, then an exception condition is raised: *invalid transaction termination (2D000)*.
- 2) All changes to GQL-data or the GQL-catalog that were made by the currently active GQL-transaction are canceled.
- 3) The currently active GQL-transaction is terminated and the current transaction is set to “not set”.

Conformance Rules

None.

8.4 <commit command>

Function

Terminate the currently active GQL-transaction with commit.

Format

<commit command> ::=
COMMIT

Syntax Rules

None.

General Rules

- 1) Case:
 - a) If the currently active GQL-transaction is part of an encompassing transaction that is controlled by an agent other than the GQL-agent, then an exception condition is raised: *invalid transaction termination (2D000)*.
 - b) If any other error preventing commitment of the GQL-transaction has occurred, then any changes to GQL-data or the GQL-catalog that were made by the currently active GQL-transaction are canceled and an exception condition is raised: *transaction rollback (40000)* with an implementation-defined (IE008) subclass value.
 - c) Otherwise, any changes to GQL-data or the GQL-catalog that were made by the currently active GQL-transaction are eligible to be perceived by all subsequent GQL-transactions.
- 2) The currently active GQL-transaction is terminated and the current transaction is set to “not set”.

Conformance Rules

None.

9 Procedure specification

9.1 <procedure specification>

Function

Specify the procedural logic of a procedure.

Format

```

<nested procedure specification> ::=
  <left brace> <procedure specification> <right brace>

<procedure specification> ::=
  <catalog-modifying procedure specification>
  | <data-modifying procedure specification>
  | <query specification>

<catalog-modifying procedure specification> ::=
  <procedure body>

<nested data-modifying procedure specification> ::=
  <left brace> <data-modifying procedure specification> <right brace>

<data-modifying procedure specification> ::=
  <procedure body>

<nested query specification> ::=
  <left brace> <query specification> <right brace>

<query specification> ::=
  <procedure body>

```

Syntax Rules

- 1) If the <nested procedure specification> *NPS* is specified, then the declared type of *NPS* is the declared type of the <procedure specification> immediately contained in *NPS*.
- 2) If the <nested query specification> *NQS* is specified, then the declared type of *NQS* is the declared type of the <query specification> immediately contained in *NQS*.
- 3) If the <nested data-modifying procedure specification> *NDPS* is specified, then the declared type of *NDPS* is the declared type of the <data-modifying procedure specification> immediately contained in *NDPS*.
- 4) If the <procedure specification> *PS* is specified, then:
 - a) Let *PB* be the <procedure body> immediately contained in the <catalog-modifying procedure specification>, <data-modifying procedure specification>, or <query specification> immediately contained in *PS*.
 - b) Case:

ISO/IEC 39075:2024(en)
9.1 <procedure specification>

- i) If *PB* simply contains a <linear catalog-modifying statement>, then *PS* is a <catalog-modifying procedure specification>.
 - ii) If *PB* simply contains a <linear data-modifying statement>, then *PS* is a <data-modifying procedure specification>.
 - iii) Otherwise, *PS* is a <query specification>.
- 5) If the <catalog-modifying procedure specification> *CPS* is specified, then every <statement> simply contained in *CPS* shall be a <linear catalog-modifying statement>.
- 6) If the <data-modifying procedure specification> *DPS* is specified, then every <statement> *S* simply contained in *DPS* shall be either a <linear data-modifying statement> or a <composite query statement>. At least one such *S* shall be a <linear data-modifying statement>.
- 7) If the <query specification> *QS* is specified, then every <statement> simply contained in *QS* shall be a <composite query statement>.

General Rules

- 1) If a <procedure specification> *PS* is specified that is immediately contained in a <transaction activity>, then:
- a) If no GQL-transaction is active, then a new GQL-transaction *TX* is initiated and the current transaction is set to *TX*.

NOTE 172 — This determines *TX* as the currently active GQL-transaction associated with the GQL-session of the currently executing GQL-request.
 - b) If *PS* is a <catalog-modifying procedure specification> and a <data-modifying procedure specification> has already occurred in the current transaction and the GQL-implementation does not support Feature GP18, “Catalog and data statement mixing”, then an exception is raised: *invalid transaction state — catalog and data statement mixing not supported (25G02)*.
 - c) If *PS* is a <data-modifying procedure specification> and a <catalog-modifying procedure specification> has already occurred in the current transaction and the GQL-implementation does not support Feature GP18, “Catalog and data statement mixing”, then an exception is raised: *invalid transaction state — catalog and data statement mixing not supported (25G02)*.
 - d) If *PS* contains a <use graph clause> with <graph expression> *GE2* and a <use graph clause> with <graph expression> *GE1* has already occurred in the current transaction and the result of *GE2* is different from the result of *GE1* and the GQL-implementation does not support Feature GT03, “Use of multiple graphs in a transaction”, then an exception is raised: *invalid transaction state — accessing multiple graphs not supported (25G04)*.

Conformance Rules

None.

9.2 <procedure body>

Function

Specify the body of a procedure.

Format

```
<procedure body> ::=  
  [ <at schema clause> ] [ <binding variable definition block> ] <statement block>  
  
<binding variable definition block> ::=  
  <binding variable definition>...  
  
<binding variable definition> ::=  
  <graph variable definition>  
  | <binding table variable definition>  
  | <value variable definition>  
  
<statement block> ::=  
  <statement> [ <next statement>... ]  
  
<statement> ::=  
  <linear catalog-modifying statement>  
  | <linear data-modifying statement>  
  | <composite query statement>  
  
<next statement> ::=  
  NEXT [ <yield clause> ] <statement>
```

Syntax Rules

- 1) Let *PB* be the <procedure body>.
- 2) If the <binding variable definition block> *BVDBLK* is specified, then:
 - a) The declared type of the incoming working record of *BVDBLK* is the declared type of the incoming working record of *PB*.
 - b) The declared type of the incoming working table of *BVDBLK* is the material unit binding table type.
 - c) Let *BVDSEQ* be the sequence of <binding variable definition>s immediately contained in *BVDBLK*, let *N* be the number of elements of *BVDSEQ*.
 - d) The declared type of the incoming working record of the first <binding variable definition> in *BVDSEQ* is the declared type of the incoming working record of *BVDBLK*.
 - e) For *j*, $1 \text{ (one)} \leq j \leq N$, let *BVD_j* be the *j*-th element of *BVDSEQ*.
 - f) For *j*, $2 \text{ (two)} \leq j \leq N$, the declared type of the incoming working record of *BVD_j* is the declared type of the outgoing working record of *BVD_{j-1}*.
 - g) The declared type of the outgoing working record of *BVDBLK* is the declared type of the outgoing working record of *BVD_N*.
 - h) The declared type of the outgoing working table of *BVDBLK* is the declared type of the incoming working table of *BVDBLK*.

- 3) Let *SBLK* be the <statement block>.
- 4) If *SBLK* directly contains a <linear catalog-modifying statement>, then *SBLK* shall directly contain at most one <statement>.
- 5) If *SBLK* directly contains a <focused linear query statement> or a <focused linear data-modifying statement>, then *SBLK* shall not directly contain an <ambient linear query statement> or an <ambient linear data-modifying statement>.

NOTE 173 — As a consequence of this rule, focused statements and ambient statement are mutually exclusive within a <statement block>, so that the following is true as well: If *SBLK* directly contains an <ambient linear query statement> or an <ambient linear data-modifying statement>, then *SBLK* does not directly contain a <focused linear query statement> or a <focused linear data-modifying statement>.

- 6) If *SBLK* directly contains a <linear query statement> that is a <select statement>, then every <linear query statement> directly contained in *SBLK* shall be a <select statement>.
- 7) If *SBLK* directly contains a <linear query statement> that contains a <primitive result statement>, then every <linear query statement> directly contained in *SBLK* shall contain a <primitive result statement>.
- 8) Let *TSTMSEQ* be the sequence of <next statement>s immediately contained in *SBLK*, let *M* be the number of elements of *TSTMSEQ*.
- 9) For *j*, $0 \text{ (zero)} \leq j \leq M$, let the <statement>s *STM_j* directly contained in *SBLK* be defined as follows:
 - a) *STM₀* is the <statement> immediately contained in *SBLK*.
 - b) For *j*, $1 \text{ (one)} \leq j \leq M$, *STM_j* is the <statement> contained in the *j*-th element of *TSTMSEQ*.
- 10) The declared type of incoming working record of *STM₀* is defined as follows.

Case:

 - a) If the <binding variable definition block> *BVDBLK* was specified, then the declared type of the incoming working record of *STM₀* is the declared type of the outgoing working record of *BVDBLK*.
 - b) Otherwise, the declared type of incoming working record of *STM₀* is the declared type of the incoming working record of *PB*.
- 11) The declared type of the incoming working table of *STM₀* is the declared type of the incoming working table of *PB*.
- 12) For *j*, $1 \text{ (one)} \leq j \leq M$:
 - a) The declared type of the incoming working record of *STM_j* is the declared type of the outgoing working record of *STM_{j-1}*.
 - b) The declared type of the incoming working table of *STM_j* is defined as follows.

Case:

- i) If the declared type of *STM_{j-1}* is a binding table type *BTT*, then

Case:

- 1) If the <next statement> that directly contains *STM_j* also directly contains a <yield clause> *YC*, then:

- A) The declared type of the incoming working table of *YC* is *BTT*.

- B) The declared type of the incoming working table of STM_j is the declared type of YC .
 - 2) Otherwise, the declared type of the incoming working table of STM_j is BTT .
 - ii) Otherwise, the declared type of the incoming working table of STM_j is the material unit binding table type.
 - c) The record type of the declared type of the incoming working table of STM_j and the declared type of the incoming working record of STM_j shall be field name-disjoint.
- 13) The declared type of PB is the declared type of STM_M .

General Rules

- 1) If the <binding variable definition block> $BVDBLK$ is specified, then:
- a) Let $CONTEXT$ be a new child execution context.
 - b) For j , $1 \text{ (one)} \leq j \leq N$,
Case:
 - i) If BVD_j is a <graph variable definition>, then the General Rules of Subclause 10.1, "<graph variable definition>" are applied in $CONTEXT$.
 - ii) If BVD_j is a <binding table variable definition>, then the General Rules of Subclause 10.2, "<binding table variable definition>" are applied in $CONTEXT$.
 - iii) Otherwise, BVD_j is a <value variable definition> and the General Rules of Subclause 10.3, "<value variable definition>" are applied in $CONTEXT$.
 - c) The current working record is set to the working record of $CONTEXT$.
- 2) The General Rules of STM_0 are applied.
- NOTE 174 — Not all statements set a result; instead they possibly modify the current execution context. A result is set by <primitive result statement>s and by <simple data-modifying statement>s to determine the execution outcome of a GQL-procedure or a top-level <statement>.
- 3) For j , $1 \text{ (one)} \leq j \leq M$:
- a) Case:
 - i) If the current execution outcome has a binding table result BTR , then the current working table is set to BTR .
 - ii) Otherwise, the current working table is set to a new unit binding table.
 - b) If the <next statement> that directly contains STM_j also directly contains a <yield clause> YC , then:
 - i) The General Rules of Subclause 16.14, "<yield clause>", are applied; let $YIELD$ be the result of YC after the application of those General Rules.
 - ii) The current working table is set to $YIELD$.
 - c) The General Rules of STM_j are applied.

NOTE 175 — Not all statements set a result; instead they possibly modify the current execution context. A result is set by <primitive result statement>s and by <simple data-modifying statement>s to determine the execution outcome of a GQL-procedure or a top-level <statement>.

- 4) The outcome of the application of these General Rules is the current execution outcome.

Conformance Rules

- 1) Without Feature GP16, “AT schema clause”, in conforming GQL language, a <procedure body> shall not contain an <at schema clause>.
- 2) Without Feature GP17, “Binding variable definition block”, in conforming GQL language, a <procedure body> shall not contain a <binding variable definition block>.
- 3) Without Feature GP11, “Procedure-local graph variable definitions”, in conforming GQL language, a <binding variable definition> shall not contain a <graph variable definition>.
- 4) Without Feature GP08, “Procedure-local binding table variable definitions”, in conforming GQL language, a <binding variable definition> shall not contain a <binding table variable definition>.
- 5) Without Feature GP05, “Procedure-local value variable definitions”, in conforming GQL language, a <binding variable definition> shall not contain a <value variable definition>.
- 6) Without Feature GQ20, “Advanced linear composition with NEXT”, in conforming GQL language, a <procedure body> shall not contain a <next statement>.

10 Variable definitions

10.1 <graph variable definition>

Function

Define graph variables.

Format

```
<graph variable definition> ::=
  [ PROPERTY ] GRAPH <binding variable> <opt typed graph initializer>

<opt typed graph initializer> ::=
  [ [ <typed> ] <graph reference value type> ] <graph initializer>

<graph initializer> ::=
  <equals operator> <graph expression>
```

Syntax Rules

- 1) Let *GVD* be the <graph variable definition>.
- 2) Let *GN* be the name of the <binding variable> simply contained in *GVD*.
- 3) Let *OTGI* be the <opt typed graph initializer> immediately contained in *GVD*.
- 4) Let *GI* be the <graph initializer> immediately contained in *OTGI*.
- 5) The declared type of *GI* is the declared type of the <graph expression> immediately contained in *GI*.
- 6) Let *WRT* be the declared type of the incoming working record of *GVD*.
- 7) *WRT* shall not have a field type whose name is *GN*.
- 8) Let the graph reference value type *GT* be defined as follows.
 - a) Let *GIT* be the declared type of *GI*.
 - b) Case:
 - i) If *GVD* simply contains a <graph reference value type> *OGT* without an intervening instance of <graph initializer>, then:
 - 1) *GIT* shall be assignable to the graph reference value type specified by *OGT*.
 - 2) *GT* is the graph reference value type specified by *OGT*.
 - ii) Otherwise, *GT* is *GIT*.
- 9) The declared type of the outgoing working record of *GVD* is a record type comprising all the field types of *WRT* and one additional field type with name *GN* and value type *GT*.

General Rules

- 1) The result of *GI* is the result of the <graph expression> immediately contained in *GI*.
- 2) Let *ROGI* be the result of *GI*.
- 3) A field with name *GN* and value assigned from *ROGI* is added to the current working record.

Conformance Rules

- 1) Without Feature GP12, “Procedure-local graph variable definitions: graph variables based on simple expressions or references”, conforming GQL language shall not contain a <graph variable definition> that contains a <procedure body>.
- 2) Without Feature GP13, “Procedure-local graph variable definitions: graph variables based on sub-queries”, conforming GQL language shall not contain a <graph variable definition> that simply contains a <graph expression> that does not conform to <value specification> or is a <graph reference>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

10.2 <binding table variable definition>

Function

Define binding table variables.

Format

```
<binding table variable definition> ::=  
  [ BINDING ] TABLE <binding variable> <opt typed binding table initializer>  
<opt typed binding table initializer> ::=  
  [ [ <typed> ] <binding table reference value type> ] <binding table initializer>  
<binding table initializer> ::=  
  <equals operator> <binding table expression>
```

Syntax Rules

- 1) Let *BTVD* be the <binding table variable definition>.
- 2) Let *BTN* be the name of the <binding variable> simply contained in *BTVD*.
- 3) Let *OTBTI* be the <opt typed binding table initializer> immediately contained in *BTVD*.
- 4) Let *BTI* be the <binding table initializer> immediately contained in *OTBTI*.
- 5) The declared type of *BTI* is the declared type of the <binding table expression> immediately contained in *BTI*.
- 6) Let *WRT* be the declared type of the incoming working record of *BTVD*.
- 7) *WRT* shall not have a field type whose name is *BTN*.
- 8) Let the binding table reference value type *BTT* be defined as follows.
 - a) Let *BTIT* be the declared type of *BTI*.
 - b) Case:
 - i) If *BTVD* simply contains a <binding table reference value type> *OBTT* without an intervening instance of <binding table initializer>, then:
 - 1) *BTIT* shall be assignable to the binding table reference value type specified by *OBTT*.
 - 2) *BTT* is the binding table reference value type specified by *OBTT*.
 - ii) Otherwise, *BTT* is *BTIT*.
- 9) The declared type of the outgoing working record of *BTVD* is a record type comprising all the field types of *WRT* and one additional field type with name *BTN* and value type *BTT*.

General Rules

- 1) The result of *BTI* is the result of the <binding table expression> immediately contained in *BTI*.
- 2) Let *ROBTI* be the result of *BTI*.

- 3) A field with name *BTN* and value assigned from *ROBTI* is added to the current working record.

Conformance Rules

- 1) Without Feature GP10, “Procedure-local binding table variable definitions: binding table variables based on subqueries”, conforming GQL language shall not contain a <binding table variable definition> that contains a <procedure body>.
- 2) Without Feature GP09, “Procedure-local binding table variable definitions: binding table variables based on simple expressions or references”, conforming GQL language shall not contain a <binding table variable definition> that simply contains a <binding table expression> that does not conform to <value specification> or is a <binding table reference>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

10.3 <value variable definition>

Function

Define value variables.

Format

```
<value variable definition> ::=  
  VALUE <binding variable> <opt typed value initializer>  
  
<opt typed value initializer> ::=  
  [ [ <typed> ] <value type> ] <value initializer>  
  
<value initializer> ::=  
  <equals operator> <value expression>
```

Syntax Rules

- 1) Let *VVD* be the <value variable definition>.
- 2) Let *VN* be the name of the <binding variable> immediately contained in *VVD*.
- 3) Let *OTVI* be the <opt typed value initializer> immediately contained in *VVD*.
- 4) Let *VI* be the <value initializer> immediately contained in *OTVI*.
- 5) The declared type of *VI* is the declared type of the <value expression> immediately contained in *VI*.
- 6) Let *WRT* be the declared type of the incoming working record of *VVD*.
- 7) *WRT* shall not have a field type whose name is *VN*.
- 8) Let the value type *VT* be defined as follows.
Case:
 - a) Let *VIT* be the declared type of *VI*.
 - b) Case:
 - i) If *OTVI* immediately contains a <value type> *OVT*, then:
 - 1) *VIT* shall be assignable to the value type specified by *OVT*.
 - 2) *VT* is the value type specified by *OVT*.
 - ii) Otherwise, *VT* is *VIT*.
- 9) The declared type of the outgoing working record of *VVD* is a record type comprising all the field types of *WRT* and one additional field type with name *VN* and value type *VT*.

General Rules

- 1) The result of *VI* is the result of the <value expression> immediately contained in *VI*.
- 2) Let *ROVI* be the result of *VI*.
- 3) A field with name *VN* and value assigned from *ROVI* is added to the current working record.

Conformance Rules

- 1) Without Feature GP07, “Procedure-local value variable definitions: value variable based on subqueries”, conforming GQL language shall not contain a <value variable definition> that contains a <procedure body>.
- 2) Without Feature GP06, “Procedure-local value variable definitions: value variables based on simple expressions”, conforming GQL language shall not contain a <value variable definition> that simply contains a <value expression> that does not conform to <value specification>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

11 Object expressions

11.1 <graph expression>

Function

Specify a graph reference value.

Format

```
<graph expression> ::=
  <object expression primary>
  | <graph reference>
  | <object name or binding variable>
  | <current graph>

<current graph> ::=
  CURRENT_PROPERTY_GRAPH | CURRENT_GRAPH
```

Syntax Rules

- 1) Let *GE* be the <graph expression>.
- 2) If *GE* is an <object name or binding variable> that is a <regular identifier> *RI*, then

Case:

 - a) If *RI* is a valid <binding variable reference> whose incoming working record's declared type is the declared type of the incoming working record of *GE*, then *GE* is effectively replaced by the <object expression primary>:

VARIABLE *RI*

NOTE 176 — *RI* is only a valid <binding variable reference>, if a binding variable referenced by *RI* is available to *GE*, i.e., if the declared type of the incoming working record of *GE* has a field type whose name is the name of the referenced binding variable of *RI*, cf. Subclause 20.12, "<binding variable reference>".
 - b) Otherwise, *GE* is effectively replaced by the <graph reference>:

./*RI*

NOTE 177 — This syntax transformation removes <object name or binding variable> as an alternative of <graph expression>, so that this alternative has not to be considered subsequently.
- 3) The declared type of *GE* is defined as follows.

Case:

 - a) If *GE* implicitly or explicitly specifies VARIABLE, then:
 - i) Let *VEP* be the <value expression primary> simply contained in *GE*.
 - ii) The declared type of *VEP* shall be a material graph reference value type.

- iii) The declared type of *GE* is the declared type of *VEP*.
- b) If *GE* is a <graph reference> *GR* that identifies a graph *G*, then:
Case:
 - i) If *G* has a constraining graph type *CGT*, then the declared type of *GE* is the graph reference value type whose constraining GQL-object type is *CGT*.
 - ii) Otherwise, the declared type of *GE* is the open graph reference value type.
- c) Otherwise, *GE* is a <current graph> and
 - i) The current working graph site of *GE* shall not be “omitted”.
 - ii) Let *CWGS* be the current working graph site of *GE*.
 - iii) The declared type of *GE* is the declared type of *CWGS*.

General Rules

- 1) The result of *GE* is the graph reference value *GRV* defined as follows.
Case:
 - a) If *GE* implicitly or explicitly specifies *VARIABLE*, then *GRV* is the result of the <value expression primary> simply contained in *GE*.
 - b) If *GE* simply contains the <graph reference> *GR*, then *GRV* is a graph reference value whose referent is the graph identified by *GR*.
 - c) Otherwise, *GE* is a <current graph> and *GRV* is the graph reference value occupying *CWGS*.

Conformance Rules

- 1) Without Feature GV60, “Graph reference value types”, in conforming GQL language, a <graph expression> *GE* shall not be an <object name or binding variable> that is a <regular identifier> that also is a valid <binding variable reference> whose incoming working record’s declared type is the declared type of the incoming working record of *GE*.

NOTE 178 — Without Feature GV60, “Graph reference value types”, <graph expression> is limited to references to graphs in the catalog and the graph referenced by the current working graph site of the <graph expression>.

11.2 <binding table expression>

Function

Specify a binding table reference value.

Format

```
<binding table expression> ::=  
    <nested binding table query specification>  
    | <object expression primary>  
    | <binding table reference>  
    | <object name or binding variable>  
  
<nested binding table query specification> ::=  
    <nested query specification>
```

Syntax Rules

- 1) Let *BTE* be the <binding table expression>.
- 2) The declared type of a <nested binding table query specification> shall be a binding table type or a binding table reference value type.
- 3) If *BTE* is an <object name or binding variable> that is a <regular identifier> *RI*, then

Case:

- a) If *RI* is a valid <binding variable reference> whose incoming working record's declared type is the declared type of the incoming working record of *BTE*, then *BTE* is effectively replaced by the <object expression primary>:

VARIABLE *RI*

NOTE 179 — *RI* is only a valid <binding variable reference>, if a binding variable referenced by *RI* is available to *BTE*, i.e., if the declared type of the incoming working record of *BTE* has a field type whose name is the name of the referenced binding variable of *RI*, cf. Subclause 20.12, "<binding variable reference>".

- b) Otherwise, *BTE* is effectively replaced by the <binding table reference>:

./*RI*

NOTE 180 — This syntax transformation removes <object name or binding variable> as an alternative of <binding table expression>, so that this alternative has not to be considered subsequently.

- 4) The declared type of *BTE* is defined as follows.

Case:

- a) If *BTE* is a <nested binding table query specification> *NBTQS*, then the declared type of *BTE* is the declared type of *NBTQS*.
- b) If *BTE* implicitly or explicitly specifies VARIABLE, then:
 - i) Let *VEP* be the <value expression primary> simply contained in *BTE*.
 - ii) The declared type of *VEP* shall be a material binding table reference value type.
 - iii) The declared type of *BTE* is the declared type of *VEP*.

- c) If *BTE* is a <binding table reference> *BTR* that identifies a binding table *BT*, then the declared type of *BTE* is a material binding table reference value type whose constraining GQL-object type is the binding table type of *BT*.

General Rules

- 1) The result of *BTE* is the binding table reference value *BTRV* defined as follows.

Case:

- a) If *BTE* simply contains the <nested binding table query specification> *NBTQS*, then

Case:

- i) If the result of the <nested query specification> immediately contained in *NBTQS* is a binding table reference value *V*, then *BTRV* is *V*.
- ii) Otherwise, *BTRV* is a binding table reference value whose referent is the binding table that is the result of the <nested query specification> immediately contained in *NBTQS*.
- b) If *BTE* implicitly or explicitly specifies *VARIABLE*, then *BTRV* is the result of the <value expression primary> simply contained in *BTE*.
- c) If *BTE* simply contains the <binding table reference> *BTR*, then *BTRV* is a binding table reference value whose referent is the binding table identified by *BTR*.

Conformance Rules

- 1) Without Feature GV61, “Binding table reference value types”, in conforming GQL language, a <binding table expression> *BTE* shall not be a <nested binding table query specification> and shall not be an <object name or binding variable> that is a <regular identifier> that also is a valid <binding variable reference> whose incoming working record’s declared type is the declared type of the incoming working record of *BTE*.

NOTE 181 — Without Feature GV61, “Binding table reference value types”, <binding table expression> is limited to references to binding tables in the GQL-catalog.

11.3 <object expression primary>

Function

Specify an <object expression primary>.

Format

```
<object expression primary> ::=  
    VARIABLE <value expression primary>  
    | <parenthesized value expression>  
    | <non-parenthesized value expression primary special case>
```

Syntax Rules

- 1) If an <object expression primary> *OEP* is specified that does not specify *VARIABLE*, then *OEP* is effectively replaced by:

VARIABLE OEP

NOTE 182 — This syntax transformation removes <parenthesized value expression> and <non-parenthesized value expression primary special case> as alternatives of <object expression primary>, so that these alternatives do not have to be considered subsequently.

General Rules

None.

Conformance Rules

None.

12 Catalog-modifying statements

12.1 <linear catalog-modifying statement>

Function

Specify a linear composition of <simple catalog-modifying statement>s.

Format

```
<linear catalog-modifying statement> ::=
  <simple catalog-modifying statement>...

<simple catalog-modifying statement> ::=
  <primitive catalog-modifying statement>
  | <call catalog-modifying procedure statement>

<primitive catalog-modifying statement> ::=
  <create schema statement>
  | <drop schema statement>
  | <create graph statement>
  | <drop graph statement>
  | <create graph type statement>
  | <drop graph type statement>
```

Syntax Rules

- 1) Let *LCMS* be the <linear catalog-modifying statement>.
- 2) The declared type of the outgoing working record of *LCMS* is the declared type of the incoming working record of *LCMS*.
- 3) The declared type of the outgoing working table of *LCMS* is the declared type of the incoming working table of *LCMS*.
- 4) The declared type of *LCMS* is the empty type.

General Rules

- 1) If the current transaction access mode is READ ONLY, then an exception condition is raised: *invalid transaction state — read-only GQL-transaction (25G03)*.
- 2) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

None.

12.2 <create schema statement>

Function

Create a GQL-schema.

Format

```
<create schema statement> ::=  
  CREATE SCHEMA [ IF NOT EXISTS ] <catalog schema parent and name>
```

Syntax Rules

- 1) Let *CSPN* be the <catalog schema parent and name>.
- 2) Let *ADP* be the <absolute directory path> immediately contained in *CSPN*. *ADP* shall identify a GQL-directory.
- 3) Let *PARENT* be the GQL-directory identified by *ADP*.
- 4) Let *SN* be the <schema name> immediately contained in *CSPN*.
- 5) If IF NOT EXISTS is not specified, then *SN* shall not identify an existing GQL-schema descriptor in *PARENT*.

General Rules

- 1) If IF NOT EXISTS is specified and *SN* identifies an existing GQL-schema descriptor in *PARENT*, then no further General Rules of this Subclause are applied.
- 2) Let *S* be a new GQL-schema.
- 3) The associated GQL-schema descriptor *SD* of *S* is created that comprises:
 - a) The session authorization identifier as the owner of *S*.
 - b) An empty set as the set of named subobjects of *S*.
- 4) *SD* is inserted into *PARENT* with the name *SN*.

Conformance Rules

- 1) Without Feature GC01, "Graph schema management", conforming GQL language shall not contain a <create schema statement>.
- 2) Without Feature GC02, "Graph schema management: IF [NOT] EXISTS", conforming GQL language shall not contain a <create schema statement> that includes IF NOT EXISTS.

12.3 <drop schema statement>

Function

Destroy a GQL-schema.

Format

```
<drop schema statement> ::=  
  DROP SCHEMA [ IF EXISTS ] <catalog schema parent and name>
```

Syntax Rules

- 1) Let *DSS* be <drop schema statement>.
- 2) Let *CSPN* be the <catalog schema parent and name>.
- 3) Let *ADP* be the <absolute directory path> immediately contained in *CSPN*. *ADP* shall identify a GQL-directory.
- 4) Let *PARENT* be the GQL-directory identified by *ADP*.
- 5) Let *SN* be the <schema name> immediately contained in *CSPN*.
- 6) If IF EXISTS is not specified, then *SN* shall identify an existing GQL-schema descriptor in *PARENT*.
- 7) If *SN* identifies an existing GQL-schema descriptor *SD* in *PARENT*, then:
 - a) If the current working schema of *DSS* is defined, then *SD* shall not be the descriptor of the current working schema of *DSS*.
 - b) *SD* shall not contain any catalog object descriptors.

General Rules

- 1) If IF EXISTS is specified and *SN* does not identify an existing GQL-schema descriptor in *PARENT*, then no further General Rules of this Subclause are applied.
- 2) *SD* is removed from *PARENT* under the name *SN* and *SD* is destroyed.

Conformance Rules

- 1) Without Feature GC01, "Graph schema management", conforming GQL language shall not contain a <drop schema statement>.
- 2) Without Feature GC02, "Graph schema management: IF [NOT] EXISTS", conforming GQL language shall not contain a <drop schema statement> that includes IF EXISTS.

12.4 <create graph statement>

Function

Create a graph.

Format

```
<create graph statement> ::=  
  CREATE { [ PROPERTY ] GRAPH [ IF NOT EXISTS ] | OR REPLACE [ PROPERTY ] GRAPH }  
    <catalog graph parent and name> { <open graph type> | <of graph type> }  
    [ <graph source> ]  
  
<open graph type> ::=  
  [ <typed> ] ANY [ [ PROPERTY ] GRAPH ]  
  
<of graph type> ::=  
  <graph type like graph>  
  | [ <typed> ] <graph type reference>  
  | [ <typed> ] [ [ PROPERTY ] GRAPH ] <nested graph type specification>  
  
<graph type like graph> ::=  
  LIKE <graph expression>  
  
<graph source> ::=  
  AS COPY OF <graph expression>
```

Syntax Rules

- 1) Let *CGS* be the <create graph statement>.
- 2) Let *CGPN* be the <catalog graph parent and name> immediately contained in *CGS*.
- 3) Let *COPR* be the explicit or implicit <catalog object parent reference> immediately contained in *CGPN*. *COPR* shall identify a GQL-schema.
- 4) Let *SCHEMA* be the GQL-schema identified by *COPR*.
- 5) Let *GN* be the <graph name> immediately contained in *CGPN*.
- 6) If *CGS* does not immediately contain IF NOT EXISTS or OR REPLACE, then *CGPN* shall not identify an existing graph.
- 7) If *CGS* immediately contains an <of graph type> that immediately contains the <graph type like graph> that immediately contains the <graph expression> *OGTGE*, then:
 - a) The declared type of the incoming working record of *OGTGE* is the declared type of the incoming working record of *CGS*.
 - b) The declared type of the incoming working table of *OGTGE* is the material unit binding table type.
- 8) If *CGS* immediately contains a <graph source> that immediately contains the <graph expression> *GSGE*, then:
 - a) The declared type of the incoming working record of *GSGE* is the declared type of the incoming working record of *CGS*.

- b) The declared type of the incoming working table of *GSGE* is the material unit binding table type.

General Rules

- 1) If IF NOT EXISTS is specified and *CGPN* identifies an existing graph, then no further General Rules of this Subclause are applied.
- 2) If *CGS* immediately contains OR REPLACE and *CGPN* identifies an existing graph, then the following <drop graph statement> is effectively executed:

DROP GRAPH *CGPN*

- 3) Let *NGTN* be the name of a new system-generated identifier that is also a <graph type name> that does not identify an existing catalog object in *SCHEMA*.

NOTE 183 — See [Syntax Rule 23](#)) of Subclause 21.3, “<token>, <separator>, and <identifier>” for detailed provisions regarding the construction of system-generated regular identifiers.

- 4) Let *NCGTPN* be the <catalog graph type parent and name>:

COPR *NGTN*

- 5) If *CGS* immediately contains an <of graph type> that immediately contains the <graph type like graph> *GTLG*, then the following <create graph type statement> is effectively executed:

CREATE GRAPH TYPE *NCGTPN* *GTLG*

- 6) If *CGS* immediately contains an <of graph type> that immediately contains the <nested graph type specification> *GTS*, then the following <create graph type statement> is effectively executed:

CREATE GRAPH TYPE *NCGTPN* *GTS*

- 7) If *CGS* immediately contains a <graph source> that immediately contains the <graph expression> *GSGE*, then:

- a) Let *SGRV* be the result of evaluating *GSGE* in a new child execution context.
- b) If the referent of *SGRV* has a constraining graph type and *CGS* does not immediately contain an <open graph type>, then the following <create graph type statement> is effectively executed:

CREATE GRAPH TYPE *NCGTPN* LIKE *GSGE*

- 8) Let *G* be a new empty graph.
- 9) Let the (possibly “omitted”) graph type *GT* be defined as follows.

Case:

- a) If *CGS* immediately contains an <open graph type>, then *GT* is “omitted”.
 - b) If *CGS* immediately contains an <of graph type> that immediately contains the <graph type reference> *GTR*, then *GT* is the graph type identified by *GTR*.
 - c) Otherwise, *GT* is the graph type identified by *NCGTPN*.
- 10) If *CGS* immediately contains a <graph source>, then *G* is populated as follows:
 - a) *SGRV* is known to be defined.
 - b) Let *SG* be the referent of *SGRV*.

- c) If *GT* is defined and *SG* is not of graph type *GT*, then an exception condition is raised: *graph type violation (G2000)*.
 - d) Copies of all nodes and edges in *SG* are inserted into *G*. Each copy of a node and each copy of an edge is associated with a new global object identifier.
- 11) The associated graph descriptor *GD* of *G* is created such that the constraining graph type of *GD* is defined as follows.
- Case:
- a) If *GT* is not “omitted”, then the constraining graph type of *GD* is *GT*.
 - b) Otherwise, *GD* has no constraining graph type.
- 12) *GD* is inserted into *SCHEMA* with the name *GN*.

Conformance Rules

- 1) Without Feature GC04, “Graph management”, conforming GQL language shall not contain a <create graph statement>.
- 2) Without Feature GC05, “Graph management: IF [NOT] EXISTS”, conforming GQL language shall not contain a <create graph statement> that includes IF NOT EXISTS.
- 3) Without Feature GG01, “Graph with an open graph type”, in conforming GQL language, a <create graph statement> shall not contain an <open graph type>.
- 4) Without Feature GG02, “Graph with a closed graph type”, in conforming GQL language, a <create graph statement> shall not contain an <of graph type>.
- 5) Without Feature GG03, “Graph type inline specification”, in conforming GQL language, a <create graph statement> shall not contain an <of graph type> that contains a <nested graph type specification>.
- 6) Without Feature GG04, “Graph type like a graph”, in conforming GQL language, a <create graph statement> shall not contain an <of graph type> that contains a <graph type like graph>.
- 7) Without Feature GG05, “Graph from a graph source”, in conforming GQL language, a <create graph statement> shall not contain a <graph source>.

12.5 <drop graph statement>

Function

Destroy a graph.

Format

```
<drop graph statement> ::=  
  DROP [ PROPERTY ] GRAPH [ IF EXISTS ] <catalog graph parent and name>
```

Syntax Rules

- 1) Let *DGS* be the <drop graph statement> and let *CGPN* be the <catalog graph parent and name>.
- 2) Let *COPR* be the explicit or implicit <catalog object parent reference> immediately contained in *CGPN*. *COPR* shall identify a GQL-schema.
- 3) Let *SCHEMA* be the GQL-schema identified by *COPR*.
- 4) Let *GN* be the <graph name> immediately contained in *CGPN*.
- 5) If *DGS* does not immediately contain IF EXISTS, then *CGPN* shall identify an existing graph.

General Rules

- 1) If *DGS* immediately contains IF EXISTS and *CGPN* does not identify an existing graph, then a completion condition is raised: *warning — graph does not exist (01G03)* and no further General Rules of this Subclause are applied.
- 2) Let *G* be the graph identified by *CGPN* and let *GD* be the graph descriptor of *G*.
- 3) Let *GT* be the graph type of *G*.
- 4) *GD* is removed from *SCHEMA* under the name *GN* and *GD* is destroyed.
- 5) If the name of *GT* is a system-generated name and *GT* is not the graph type of any other graph in the GQL-catalog, then the graph type descriptor of *GT* is destroyed.

NOTE 184 — See [Syntax Rule 23](#) of [Subclause 21.3](#), “<token>, <separator>, and <identifier>” for detailed provisions regarding the construction of system-generated regular identifiers.

Conformance Rules

- 1) Without Feature GC04, “Graph management”, conforming GQL language shall not contain a <drop graph statement>.
- 2) Without Feature GC05, “Graph management: IF [NOT] EXISTS”, conforming GQL language shall not contain a <drop graph statement> that includes IF EXISTS.

12.6 <create graph type statement>

Function

Create a graph type.

Format

```
<create graph type statement> ::=  
  CREATE  
    { [ PROPERTY ] GRAPH TYPE [ IF NOT EXISTS ] | OR REPLACE [ PROPERTY ] GRAPH TYPE }  
    <catalog graph type parent and name> <graph type source>  
  
<graph type source> ::=  
  [ AS ] <copy of graph type>  
  | <graph type like graph>  
  | [ AS ] <nested graph type specification>  
  
<copy of graph type> ::=  
  COPY OF { <graph type reference> | <external object reference> }
```

Syntax Rules

- 1) Let *CGTS* be the <create graph type statement> and let *CGTPN* be the <catalog graph type parent and name> immediately contained in *CGTS*.
- 2) Let *COPR* be the explicit or implicit <catalog object parent reference> immediately contained in *CGTPN*. *COPR* shall identify a GQL-schema.
- 3) Let *SCHEMA* be the GQL-schema identified by *COPR*.
- 4) Let *GTN* be the <graph type name> immediately contained in *CGTPN*.
- 5) If *CGTS* does not immediately contain IF NOT EXISTS or OR REPLACE, then *CGTPN* shall not identify an existing graph type.
- 6) Let *GTS* be the <graph type source> immediately contained in *GTS*.
- 7) Let the (possibly “omitted”) graph type *OGT* be defined as follows.

Case:

- a) If *GTS* immediately contains the <copy of graph type> *COGT*, then *OGT* is the graph type identified by the <graph type reference> or the <external object reference> immediately contained in *COGT*.
- b) If *GTS* immediately contains a <graph type like graph> *GTLG*, then:
 - i) Let *GTSGE* be the <graph expression> immediately contained in *GTLG*.
 - ii) The declared type of the incoming working record of *GTSGE* is the declared type of the incoming working record of *CGTS*.
 - iii) The declared type of the incoming working table of *GTSGE* is the material unit binding table type.
 - iv) The declared type of *GTSGE* shall have a constraining GQL-object type.
 - v) *OGT* is the constraining GQL-object type of the declared type of *GTSGE*.

- c) Otherwise, *OGT* is “omitted”.

General Rules

- 1) If IF NOT EXISTS is specified and *CGTPN* identifies an existing graph type, then no further General Rules of this Subclause are applied.
- 2) If *CGTS* immediately contains OR REPLACE and *CGTPN* identifies an existing graph type, then the following <drop graph type statement> is effectively executed:

DROP GRAPH TYPE *CGTPN*

- 3) Let the (possibly “omitted”) graph type *GT* and the graph type descriptor *GTD* be defined as follows.
Case:

- a) If *OGT* is not “omitted”, then *GT* is the graph type described by the graph type descriptor *GTD* comprising:

- i) The declared name of the primary base type of all graph types (GRAPH DATA).

NOTE 185 — See Subclause 4.13.2.2, “Graph type descriptors”.

- ii) The preferred name of *OGT* as the preferred name.
- iii) A copy of the node type set of *OGT* as the node type set.
- iv) A copy of the edge type set of *OGT* as the edge type set.
- v) A copy of the node type key label set dictionary of *OGT* as the node type key label set dictionary.
- vi) A copy of the edge type key label set dictionary of *OGT* as the edge type key label set dictionary.

- b) Otherwise, *GT* is the graph type described by the graph type descriptor *GTD* that is defined by the <nested graph type specification> immediately contained in *GTS*.

- 4) The associated graph type descriptor *GTD* of *GT* is created.

- 5) *GTD* is inserted into *SCHEMA* with the name *GTN*.

Conformance Rules

- 1) Without Feature GG02, “Graph with a closed graph type”, conforming GQL language shall not contain a <create graph type statement>.
- 2) Without Feature GC03, “Graph type: IF [NOT] EXISTS”, conforming GQL language shall not contain a <create graph type statement> that includes IF NOT EXISTS.

12.7 <drop graph type statement>

Function

Destroy a graph type.

Format

<drop graph type statement> ::=
DROP [PROPERTY] GRAPH TYPE [IF EXISTS] <catalog graph type parent and name>

Syntax Rules

- 1) Let *DGTS* be the <drop graph type statement> and *CGTPN* be the <catalog graph type parent and name>.
- 2) Let *COPR* be the explicit or implicit <catalog object parent reference> immediately contained in *CGTPN*. *COPR* shall identify a GQL-schema.
- 3) Let *SCHEMA* be the GQL-schema identified by *COPR*.
- 4) Let *GTN* be the <graph type name> immediately contained in *CGTPN*.
- 5) If *DGTS* does not immediately contain IF EXISTS, then *CGTPN* shall identify an existing graph type.
- 6) If *CGTPN* identifies an existing graph type, then the graph type identified by *CGTPN* shall not be referenced by any existing graph in the GQL-catalog.

General Rules

- 1) If *DGTS* immediately contains IF EXISTS and *CGTPN* does not identify an existing graph type, then a completion condition is raised: *warning — graph type does not exist (01G04)* and no further General Rules of this Subclause are applied.
- 2) Let *GT* be the graph type identified by *CGTPN* and let *GTD* be the graph descriptor of *GT*.
- 3) *GTD* is removed from *SCHEMA* under the name *GTN* and *GTD* is destroyed.

Conformance Rules

- 1) Without Feature GG02, “Graph with a closed graph type”, conforming GQL language shall not contain a <drop graph type statement>.
- 2) Without Feature GC03, “Graph type: IF [NOT] EXISTS”, conforming GQL language shall not contain a <drop graph type statement> that includes IF EXISTS.

12.8 <call catalog-modifying procedure statement>

Function

Execute a catalog-modifying procedure.

Format

<call catalog-modifying procedure statement> ::=
 <call procedure statement>

Syntax Rules

- 1) Let *CCPS* be the <call catalog-modifying procedure statement>, let *CPS* be the <call procedure statement> immediately contained in *CCPS*, and let *PC* be the <procedure call> immediately contained in *CPS*.
- 2) If *PC* is an <inline procedure call> that immediately contains the <nested procedure specification> *PROC*, then *PROC* shall immediately contain a <catalog-modifying procedure specification>.
- 3) If *PC* is a <named procedure call> that immediately contains the <procedure reference> that identifies a procedure *PROC*, then *PROC* shall have the CATALOG PROCEDURE indication.

General Rules

None.

Conformance Rules

None.

13 Data-modifying statements

13.1 <linear data-modifying statement>

Function

Specify a linear composition of at least one <simple data-modifying statement> with <simple query statement>s and <simple data-accessing statement>s.

Format

```

<linear data-modifying statement> ::=
  <focused linear data-modifying statement>
  | <ambient linear data-modifying statement>

<focused linear data-modifying statement> ::=
  <focused linear data-modifying statement body>
  | <focused nested data-modifying procedure specification>

<focused linear data-modifying statement body> ::=
  <use graph clause> <simple linear data-accessing statement>
  [ <primitive result statement> ]

<focused nested data-modifying procedure specification> ::=
  <use graph clause> <nested data-modifying procedure specification>

<ambient linear data-modifying statement> ::=
  <ambient linear data-modifying statement body>
  | <nested data-modifying procedure specification>

<ambient linear data-modifying statement body> ::=
  <simple linear data-accessing statement> [ <primitive result statement> ]

<simple linear data-accessing statement> ::=
  <simple data-accessing statement>...

<simple data-accessing statement> ::=
  <simple query statement>
  | <simple data-modifying statement>

<simple data-modifying statement> ::=
  <primitive data-modifying statement>
  | <call data-modifying procedure statement>

<primitive data-modifying statement> ::=
  <insert statement>
  | <set statement>
  | <remove statement>
  | <delete statement>

```

Syntax Rules

- 1) Let *LDMS* be the <linear data-modifying statement>.

ISO/IEC 39075:2024(en)
13.1 <linear data-modifying statement>

- 2) If the <focused linear data-modifying statement body> does not immediately contain a <primitive result statement>, then FINISH is implicit.
- 3) If the <ambient linear data-modifying statement body> does not immediately contain a <primitive result statement>, then FINISH is implicit.
- 4) Let $STMSEQ$ be the sequence of <simple data-accessing statement>s, the <primitive result statement>, and the <nested data-modifying procedure specification> directly contained in $LDMS$. Let M be the number of elements of $STMSEQ$. For j , $1 \text{ (one)} \leq j \leq M$, let STM_j be the j -th element of $STMSEQ$.
- 5) $STMSEQ$ shall contain at least one <simple data-modifying statement>.
- 6) The declared type of the incoming working record of STM_1 is the declared type of the incoming working record of $LDMS$.
- 7) The declared type of the incoming working table of STM_1 is the declared type of the incoming working table of $LDMS$.
- 8) For $2 \text{ (two)} \leq j \leq M$:
 - a) The declared type of the incoming working record of STM_j is the declared type of the outgoing working record of STM_{j-1} .
 - b) The declared type of the incoming working table of STM_j is the declared type of the outgoing working table of STM_{j-1} .
- 9) The declared type of $LDMS$ is the declared type of STM_M .

General Rules

- 1) If the current transaction access mode is READ ONLY, then an exception condition is raised: *invalid transaction state — read-only GQL-transaction (25G03)*.

Conformance Rules

- 1) Without Feature GQ01, “USE graph clause”, conforming GQL language shall not contain a <focused linear data-modifying statement>.
- 2) Without Feature GD01, “Updatable graphs”, conforming GQL language shall not contain a <simple data-modifying statement>.

13.2 <insert statement>

Function

Insert new nodes and edges into a graph.

Format

```
<insert statement> ::=  
  INSERT <insert graph pattern>
```

Syntax Rules

- 1) Let *IS* be the <insert statement>.
- 2) Let *IGP* be the <insert graph pattern> immediately contained in *IS*.
- 3) Let *IDNSET* be the system-generated variable names of *IGP*.
NOTE 186 — See [Syntax Rule 23](#)) of [Subclause 21.3](#), “<token>, <separator>, and <identifier>” for detailed provisions regarding the construction of system-generated regular identifiers.
- 4) Let *NEWCOLS* be a set of columns determined as follows. For every <insert path pattern> *IPP* immediately contained in *IGP*, for every insert element pattern *IEP* immediately contained in *IPP* that declares an element variable *EV* with name *EVN* that is not a bound insert element pattern:
NOTE 187 — Insert element patterns and bound insert element patterns are defined in [Syntax Rule 7](#)) and [Syntax Rule 8](#)) of [Subclause 16.5](#), “<insert graph pattern>”.
 - a) Let *COLTYPE* be determined as follows.
Case:
 - i) If *IEP* is an <insert node pattern>, then *COLTYPE* is a node reference value type that is determined using an implementation-defined ([IW012](#)) mechanism such that it includes all reference values to nodes that can be bound to *EV* by *IS*.
 - ii) Otherwise, *IEP* is an <insert edge pattern> and *COLTYPE* is an edge reference value type that is determined using an implementation-defined ([IW012](#)) mechanism such that it includes all reference values to edges that can be bound to *EV* by *IS*.
 - b) Let *COL* be the column whose name is *EVN* and whose type is *COLTYPE*.
 - c) Include *COL* in *NEWCOLS*.
 - d) If *COLTYPE* is a closed graph element reference value type, then for every <property key value pair> *PKVP* contained in *IEP*:
 - i) Let *PT* be the property type in the constraining GQL-object type of *COLTYPE* identified by the <property name> immediately contained in *PKVP*.
 - ii) The Syntax Rules of [Subclause 22.10](#), “Store assignment”, are applied with a transient site of type *PT* as *TARGET* and the <value expression> immediately contained in *PKVP* as *VALUE*.
- 5) The current working graph site of *IS* shall not be “omitted”.
- 6) Let *CWGS* be the current working graph site of *IS*.

- 7) The declared type of the outgoing working table of *IS* is the material binding table type whose set of columns is the union of all columns of the declared type of incoming working table of *IS* and all columns from *NEWCOLS* without the columns identified by *IDNSET*.
- 8) The declared type of the outgoing working record of *IS* is the declared type of the incoming working record of *IS*.
- 9) The declared type of *IS* is the empty type.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *CG* be the graph referenced by *CWGS*.
- 3) Let *NEW_TABLE* be a new empty binding table whose columns are the union of all columns of the declared type of *TABLE* and all columns from *NEWCOLS*.
- 4) For every record *R* of *TABLE* in a new child execution context amended with *R*:
 - a) Let *CR* be a new record whose field types are the elements of *NEWCOLS* and whose fields are determined by processing each <insert path pattern> *IPP* immediately contained in *IGP* as follows:
 - i) For every <insert node pattern> *INP* immediately contained in *IPP* that declares an element variable *EV* with name *EVN* that is the defining insert element pattern of *EV*, and is not a bound insert element pattern:

NOTE 188 — The defining insert element patterns of an <element variable> are defined in Syntax Rule 10)b) of Subclause 16.5, "<insert graph pattern>".

- 1) Let the set of node label names *NLNS* be determined as follows.

Case:

 - A) If *INP* immediately contains a <label set specification> *LSS*, then *NLNS* is the set of label names specified by *LSS*.
 - B) Otherwise, *NLNS* is the empty set.
- 2) If the cardinality of *NLNS* is less than the implementation-defined (IL001) node label set minimum cardinality, then an exception condition is raised: *data exception — number of node labels below supported minimum (22G0N)*.
- 3) If the cardinality of *NLNS* is greater than the implementation-defined (IL001) node label set maximum cardinality, then an exception condition is raised: *data exception — number of node labels exceeds supported maximum (22G0P)*.
- 4) Let the set of node properties *NPS* be determined as follows.

Case:

 - A) If *INP* simply contains a <property key value pair list> *PKVPL*, then let *NPS* be the list of all properties computed as follows. For each <property key value pair> *PKVP* in *PKVPL*:
 - I) Let *NPROP* be a new property.
 - II) The name of *NPROP* is the name specified by the <property name> immediately contained in *PKVP*.

- III) The General Rules of Subclause 22.10, “Store assignment”, are applied with *NPROP* as *TARGET* and the result of the <value expression> immediately contained in *PKVP* as *VALUE*.
- IV) Include *NPROP* in *NPS*.
- B) Otherwise, *NPS* is the empty set.
- 5) If the cardinality of *NPS* is greater than the implementation-defined (IL002) node property set maximum cardinality, then an exception condition is raised: *data exception — number of node properties exceeds supported maximum (22G0S)*.
- 6) Schedule the insertion of a new node *NN* into *CG*.
 - NOTE 189 — Scheduled insertions of new nodes are performed by the application of General Rule 5) of this Subclause.
 - A) *NN* is associated with a new global object identifier.
 - B) The node label set of *NN* is *NLNS*.
 - C) The node property set of *NN* is *NPS*.
- 7) Let *NR* be a node reference value for *NN*. The reference value for *INP* is *NR*.
- 8) The field whose name is *EVN* and whose value is *NR* is included in *CR*.
- ii) For every <insert edge pattern> *IEP* immediately contained in *IPP* that declares an element variable *EV* with name *EVN*:
 - 1) Let the set of label names *ELNS* be defined as follows.
 - Case:
 - A) If *IEP* immediately contains a <label set specification> *LSS*, then *ELNS* is the set of label names specified by *LSS*.
 - B) Otherwise, *ELNS* is the empty set.
 - 2) If the cardinality of *ELNS* is less than the implementation-defined (IL001) edge label set minimum cardinality, then an exception condition is raised: *data exception — number of edge labels below supported minimum (22G0Q)*.
 - 3) If the cardinality of *ELNS* is greater than the implementation-defined (IL001) edge label set maximum cardinality, then an exception condition is raised: *data exception — number of edge labels exceeds supported maximum (22G0R)*.
 - 4) Let the set of edge properties *EPS* be determined as follows.
 - Case:
 - A) If *IEP* simply contains a <property key value pair list> *PKVPL*, then let *EPS* be the list of all properties computed as follows. For each <property key value pair> *PKVP* in *PKVPL*:
 - I) Let *EPROP* be a new property.
 - II) The name of *EPROP* is the name specified by the <property name> immediately contained in *PKVP*.
 - III) The General Rules of Subclause 22.10, “Store assignment”, are applied with *EPROP* as *TARGET* and the result of the <value expression> immediately contained in *PKVP* as *VALUE*.

- IV) Include *EPROP* in *EPS*.
- B) Otherwise, *EPS* is the empty set.
- 5) If the cardinality of *EPS* is greater than the implementation-defined (IL002) edge property set maximum cardinality, then an exception condition is raised: *data exception — number of edge properties exceeds supported maximum (22G0T)*.
- 6) Let the left endpoint *N_LEFT* be defined as follows:
- A) Let *INP_LEFT* be the rightmost <insert node pattern> that precedes *IEP* in *IPP*.
- B) Case:
- I) If *INP_LEFT* is a bound insert element pattern, then *N_LEFT* is the result of the <element variable> declared by *INP_LEFT*.
- NOTE 190 — Bound insert element patterns are defined in Syntax Rule 7) and Syntax Rule 8) of Subclause 16.5, “<insert graph pattern>”.
- II) Otherwise, *N_LEFT* is the reference value for *INP_LEFT*.
- NOTE 191 — The reference value for *INP_LEFT* is defined by General Rule 4)a)i)7).
- 7) Let the right endpoint *N_RIGHT* be defined as follows:
- A) Let *INP_RIGHT* be the leftmost <insert node pattern> that follows *IEP* in *IPP*.
- B) Case:
- I) If *INP_RIGHT* is a bound insert element pattern, then *N_RIGHT* is the result of the <element variable> declared by *INP_RIGHT*.
- NOTE 192 — Bound insert element patterns are defined in Syntax Rule 8) of Subclause 16.5, “<insert graph pattern>”.
- II) Otherwise, *N_RIGHT* is the reference value for *INP_RIGHT*.
- NOTE 193 — The reference value for *INP_RIGHT* is defined by General Rule 4)a)i)7).
- 8) If either *N_LEFT* or *N_RIGHT* is deleted, then an exception condition is raised: *dependent object error — endpoint node is deleted (G1002)*.
- 9) If either *N_LEFT* or *N_RIGHT* is not in *CG*, then an exception condition is raised: *dependent object error — endpoint node not in current working graph (G1003)*.
- 10) If *N_LEFT* and *N_RIGHT* are both nodes, then:
- A) Let the <edge kind> *EK* be defined as follows. If *IEP* immediately contains a <full edge undirected>, then *EK* is *UNDIRECTED*; otherwise, *EK* is *DIRECTED*.
- B) Let *N_LEFT2* and *N_RIGHT2* be defined as follows. If *EK* is *DIRECTED* and *IEP* contains a <full edge pointing left>, then *N_LEFT2* is *N_RIGHT* and *N_RIGHT2* is *N_LEFT*; otherwise, *N_LEFT2* is *N_LEFT* and *N_RIGHT2* is *N_RIGHT*.
- 11) Schedule the insertion of a new edge *NE* into *CG*.

NOTE 194 — Scheduled insertions of new edges are performed by the application of General Rule 6) of this Subclause.

- A) *NE* is associated with a new global object identifier.
- B) The edge label set of *NE* is *ELNS*.
- C) The edge property set of *NE* is *EPS*.
- D) If *EK* is DIRECTED, then *NE* is a directed edge that connects the source node *N_LEFT2* to the destination node *N_RIGHT2*; otherwise, *EK* is UNDIRECTED and *NE* is an undirected edge that connects the endpoints *N_LEFT2* and *N_RIGHT2*.

12) Let *ER* be the edge reference value for *NE*.

13) The field whose name is *EVN* and whose value is *ER* is included in *CR*.

- b) Let *NR* be a new record constructed by amending *R* with *CR*.
 - c) Append *NR* to *NEW_TABLE*.
- 5) Perform all scheduled insertions of new nodes.
 - 6) Perform all scheduled insertions of new edges.
 - 7) If *CG* has a graph type *GT* and *CG* is not of graph type *GT*, then an exception condition is raised: *graph type violation (G2000)*.
 - 8) The current working table is set to a copy of *NEW_TABLE* without the columns identified by *IDNSET*.
 - 9) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

None.

13.3 <set statement>

Function

Set graph element properties and labels.

Format

```
<set statement> ::=  
    SET <set item list>  
  
<set item list> ::=  
    <set item> [ { <comma> <set item> }... ]  
  
<set item> ::=  
    <set property item>  
    | <set all properties item>  
    | <set label item>  
  
<set property item> ::=  
    <binding variable reference> <period> <property name> <equals operator> <value expression>  
  
<set all properties item> ::=  
    <binding variable reference> <equals operator>  
    <left brace> [ <property key value pair list> ] <right brace>  
  
<set label item> ::=  
    <binding variable reference> <is or colon> <label name>
```

Syntax Rules

- 1) Let *SS* be the <set statement>.
- 2) Let *SIL* be the <set item list> immediately contained in *SS*.
- 3) For every <set item> *SI* immediately contained in *SIL*,
Case:
 - a) If *SI* immediately contains a <set property item> *SPI*, then:
 - i) Let *BVR* be the <binding variable reference> immediately contained in *SPI*.
 - ii) Let *VE* be the <value expression> immediately contained in *SPI*.
 - iii) The declared type of *BVR* shall be a graph element reference value type.
 - iv) The declared type of *VE* immediately shall be a supported property value type.
 - v) If the declared type of *BVR* is a closed graph element reference value type *ERVT*, then:
 - 1) Let *PT* be the property type in the constraining GQL-object type of *ERVT* identified by the <property name> immediately contained in *SPI*.
 - 2) The Syntax Rules of [Subclause 22.10, “Store assignment”](#), are applied with a transient site of type *PT* as *TARGET* and *VE* as *VALUE*.
 - b) If *SI* immediately contains a <set all properties item> *SAPI*, then:
 - i) Let *BVR* be the <binding variable reference> immediately contained in *SAPI*.

ISO/IEC 39075:2024(en)
13.3 <set statement>

- ii) The declared type of *BVR* shall be a graph element reference value type.
- iii) For every <property key value pair> *PKVP* immediately contained in *SAPI*:
 - 1) Let *VE* be the <value expression> immediately contained in *PKVP*.
 - 2) The declared type of *VE* shall be a supported property value type.
 - 3) If the declared type of *BVR* is a closed graph element reference value type *ERVT*, then:
 - A) Let *PT* be the property type in the constraining GQL-object type of *ERVT* identified by the <property name> immediately contained in *PKVP*.
 - B) The Syntax Rules of [Subclause 22.10, “Store assignment”](#), are applied with a transient site of type *PT* as *TARGET* and *VE* as *VALUE*.
- c) If *SI* immediately contains a <set label item> *SLI*, then the declared type of the <binding variable reference> immediately contained in *SLI* shall be a graph element reference value type.
- 4) No <set property item> immediately contained in *SIL* shall immediately contain a <binding variable reference> and <property name> that are both equivalent to the <binding variable reference> and <property name> immediately contained in a different <set property item>.
- 5) No <set all properties item> immediately contained in *SIL* shall immediately contain a <binding variable reference> that is equivalent to the <binding variable reference> of any other <set all properties item> or <set property item> in *SIL*.
- 6) The declared type of the outgoing working table of *SS* is the declared type of the incoming working table of *SS*.
- 7) The declared type of the outgoing working record of *SS* is the declared type of the incoming working record of *SS*.
- 8) The declared type of *SS* is the empty type.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *N* be the number of records of *TABLE*. Let *M* be the number of <set item>s immediately contained in *SIL*.
- 3) For each *j*-th <set item> *SI_j*, $1 \text{ (one)} \leq j \leq M$, let *P_j* be defined as follows.

Case:

 - a) If *SI_j* immediately contains a <set all properties item> *SAPI_j*, then *P_j* is the number of <property key value pair>s that are immediately contained in *SAPI_j*.
 - b) Otherwise, *P_j* is 1 (one).
- 4) For each *j*-th <set item> *SI_j*, $1 \text{ (one)} \leq j \leq M$, let the (possibly “omitted”) property name *PN_{j,k}*, $1 \text{ (one)} \leq k \leq P_j$ be defined as follows.

Case:

 - a) If *SI_j* immediately contains a <set property item> *SPI_j*, then *PN_{j,k}* is the name specified by the <property name> immediately contained in *SPI_j*.

ISO/IEC 39075:2024(en)
13.3 <set statement>

- b) If SI_j immediately contains a <set all properties item> $SAPI_j$, then $PN_{j,k}$ is the name specified by the <property name> immediately contained in the k -th <property key value pair> simply contained in $SAPI_j$.
- c) Otherwise, $PN_{j,k}$ is “omitted”.
- 5) For each j -th <set item> SI_j , $1 \text{ (one)} \leq j \leq M$, let the (possibly “omitted”) label LN_j be defined as follows.
- Case:
- a) If SI_j immediately contains a <set label item> SLI_j , then LN_j is the label specified by the <label name> that is simply contained in SLI_j .
- b) Otherwise, LN_j is “omitted”.
- 6) For each i -th record R_i , $1 \text{ (one)} \leq i \leq N$, of *TABLE*, in a new child execution context amended with R_i :
- a) For each j -th <set item> SI_j , $1 \text{ (one)} \leq j \leq M$, immediately contained in *SIL*, let GE_{ij} be the value of the <binding variable reference> simply contained in SI_j .
- b) For each j -th <set item> SI_j , $1 \text{ (one)} \leq j \leq M$, immediately contained in *SIL*, let the (possibly “omitted”) property value $PV_{ij,k}$, $1 \text{ (one)} \leq k \leq P_j$ be defined as follows.
- Case:
- i) If SI_j immediately contains a <set property item> SPI_j , then $PV_{ij,k}$ is the result of the <value expression> immediately contained in SPI_j .
- ii) If SI_j immediately contains a <set all properties item> $SAPI_j$, then $PV_{ij,k}$ is the result of the <value expression> immediately contained in the k -th <property key value pair> simply contained in $SAPI_j$.
- iii) Otherwise, $PV_{ij,k}$ is “omitted”.
- 7) If there are two or more equivalent pairs $PNGE_{ij,k}$ of property name $PN_{j,k}$ and graph element GE_{ij} , $1 \text{ (one)} \leq i \leq N$, $1 \text{ (one)} \leq j \leq M$, $1 \text{ (one)} \leq k \leq P_j$ where $PN_{j,k}$ is defined, then it is implementation-defined (IA017) which one of the following occurs:
- a) An exception condition is raised: *data exception — multiple assignments to a graph element property* (22G0M).
- b) An implementation-dependent (UV009) choice of one of the equivalent pairs $PNGE_{ij,k}$ is made and the corresponding $PV_{ij,k}$ is used to perform the assignment to GE_{ij} .
- 8) For each i -th record R_i , $1 \text{ (one)} \leq i \leq N$, of *TABLE* (using the same order as General Rule 6)), perform the following data-modifying operations in a new child execution context amended with R_i for each j -th <set item> SI_j , $1 \text{ (one)} \leq j \leq M$, immediately contained in *SIL*. If GE_{ij} is not the null value, then
- Case:
- a) If SI_j immediately contains a <set property item> SPI_j , then the General Rules of Subclause 22.10, “Store assignment”, are applied with the property $PN_{ij,1}$ of GE_{ij} as *TARGET* and $PV_{ij,1}$ as *VALUE*.
- b) If SI_j immediately contains a <set all properties item> $SAPI_j$, then:
- i) All properties of GE_{ij} are removed.

- ii) For each <property key value pair> $PKVP_k$, $1 \text{ (one)} \leq k \leq P_j$, immediately contained in $SAPI_j$, the General Rules of Subclause 22.10, “Store assignment”, are applied with the property $PN_{j,k}$ of GE_{ij} as *TARGET* and $PV_{ij,k}$ as *VALUE*.
- c) If SI_j immediately contains a <set label item>, and LN_j is not contained in the label set of GE_{ij} , then LN_j is added to the label set of GE_{ij} .
- d) If GE_{ij} is a node and the cardinality of the node label set of GE_{ij} is greater than the implementation-defined (IL001) node label set maximum cardinality, then an exception condition is raised: *data exception — number of node labels exceeds supported maximum (22G0P)*.
- e) If GE_{ij} is a node and the cardinality of the node property set of GE_{ij} is greater than the implementation-defined (IL002) node property set maximum cardinality, then an exception condition is raised: *data exception — number of node properties exceeds supported maximum (22G0S)*.
- f) If GE_{ij} is an edge and the cardinality of the edge label set of GE_{ij} is greater than the implementation-defined (IL001) edge label set maximum cardinality, then an exception condition is raised: *data exception — number of edge labels exceeds supported maximum (22G0R)*.
- g) If GE_{ij} is an edge and the cardinality of the edge property set of GE_{ij} is greater than the implementation-defined (IL002) edge property set maximum cardinality, then an exception condition is raised: *data exception — number of edge properties exceeds supported maximum (22G0T)*.
- 9) Let CG be the graph that contains GE_{ij} . If CG has a constraining graph type but CG is not of that graph type, then an exception condition is raised: *graph type violation (G2000)*.
- 10) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GD02, “Graph label set changes”, conforming GQL language shall not contain a <set item> that is a <set label item>.

13.4 <remove statement>

Function

Remove graph element properties and labels.

Format

```
<remove statement> ::=  
  REMOVE <remove item list>
```

```
<remove item list> ::=  
  <remove item> [ { <comma> <remove item> }... ]
```

```
<remove item> ::=  
  <remove property item> | <remove label item>
```

```
<remove property item> ::=  
  <binding variable reference> <period> <property name>
```

```
<remove label item> ::=  
  <binding variable reference> <is or colon> <label name>
```

Syntax Rules

- 1) Let *RS* be the <remove statement>.
- 2) Let *RIL* be the <remove item list> immediately contained in *RS*.
- 3) For every <remove item> *RI* immediately contained in *RIL*, the declared type of the <binding variable reference> immediately contained in *RI* shall be a graph element reference value type.
- 4) The declared type of the outgoing working table of *RS* is the declared type of the incoming working table of *RS*.
- 5) The declared type of the outgoing working record of *RS* is the declared type of the incoming working record of *RS*.
- 6) The declared type of *RS* is the empty type.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *N* be the number of records of *TABLE* and let *M* be the number of <remove item>s immediately contained in *RIL*.
- 3) For each *i*-th record *R_i*, $1 \text{ (one)} \leq i \leq N$, of *TABLE* in a new child execution context amended with *R_i*:
 - a) For each *j*-th <remove item> *RI_j*, $1 \text{ (one)} \leq j \leq M$, immediately contained in *RIL*, let *GE_{ij}* be the value of the <binding variable reference> simply contained in *RI_j*.
 - b) For each *j*-th <remove item> *RI_j*, $1 \text{ (one)} \leq j \leq M$, immediately contained in *RS*, if *RI_j* immediately contains a <remove label item> *RLI_j*, then let *LN_j* be the <label name> simply contained in *RLI_j*; otherwise, let *LN_j* be “omitted”.

- 4) For each i -th record R_i , $1 \text{ (one)} \leq i \leq N$, of *TABLE* (using the same order as [General Rule 3](#))), perform the following data-modifying operations in a new child execution context amended with R_i for each j -th <remove item> RI_j , $1 \text{ (one)} \leq j \leq M$, immediately contained in *RIL*.

Case:

- a) If RI_j immediately contains a <remove property item> RPI_j , then let PN_j be the <property name> immediately contained in RPI_j . If GE_{ij} is not the null value and a property PN_j is contained in the property set of GE_{ij} , then the property PN_j is removed from the property set of GE_{ij} .
- b) If RI_j immediately contains a <remove label item> and GE_{ij} is not the null value, then:
- i) If LN_j is contained in the label set of GE_{ij} , then LN_j is removed from the label set of GE_{ij} .
- ii) If GE_{ij} is a node and the cardinality of the label set of GE_{ij} is less than the implementation-defined (IL001) node label set minimum cardinality, then an exception condition is raised: *data exception — number of node labels below supported minimum (22G0N)*.
- iii) If GE_{ij} is an edge and the cardinality of the label set of GE_{ij} is less than the implementation-defined (IL001) edge label set minimum cardinality, then an exception condition is raised: *data exception — number of edge labels below supported minimum (22G0Q)*.
- c) Let CG be the graph that contains GE_{ij} . If CG has a constraining graph type but CG is not of that graph type, then an exception condition is raised: *graph type violation (G2000)*.
- 5) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GD02, “Graph label set changes”, conforming GQL language shall not contain a <remove item> that is a <remove label item>.

13.5 <delete statement>

Function

Delete graph elements.

Format

```
<delete statement> ::=  
  [ DETACH | NODETACH ] DELETE <delete item list>  
  
<delete item list> ::=  
  <delete item> [ { <comma> <delete item> }... ]  
  
<delete item> ::=  
  <value expression>
```

Syntax Rules

- 1) Let *DS* be the <delete statement>.
- 2) Let *DIL* be the <delete item list> immediately contained in *DS*.
- 3) The current working graph site of *DS* shall not be “omitted”.
- 4) Let *CWGS* be the current working graph site of *DS*.
- 5) The declared type for every <value expression> simply contained in *DIL* shall be a graph element reference value type.
- 6) If *DS* does not specify DETACH or NODETACH, then NODETACH is implicit.
- 7) The declared type of the outgoing working table of *DS* is the declared type of the incoming working table of *DS*.
- 8) The declared type of the outgoing working record of *DS* is the declared type of the incoming working record of *DS*.
- 9) The declared type of *DS* is the empty type.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *CG* be the graph referenced by *CWGS*.
- 3) Let *NODES* be an empty set. Let *EDGES* be an empty set.
- 4) For each record *R* of *TABLE* in a new child execution context amended with *R*:
 - a) For each <delete item> *DI* immediately contained in *DIL*:
 - i) Let the value *V* be the result of the <value expression> immediately contained in *DI*.
 - ii) Case:
 - 1) If *V* is a reference value to an edge *E* and *V* is not invalidated, then *E* is added to *EDGES*.

- 2) If V is a reference value to a node N and V is not invalidated, then:
 - A) N is added to $NODES$.
 - B) If DETACH is specified, then all edges connected to N are added to $EDGES$.
- 3) Otherwise, V is the null value.
- 5) For every node N in $NODES$, if any edge connected to N is not in $EDGES$, then an exception condition is raised: *dependent object error — edges still exist (G1001)*.
- 6) Let $CG1$ be a copy of CG with all members of $EDGES$ and all members of $NODES$ deleted. If CG has a graph type GT and $CG1$ is not of graph type GT , then an exception condition is raised: *graph type violation (G2000)*.

NOTE 195 — This document does not currently specify a situation in which the above exception would be triggered. However, it is included as it is expected that implementation-defined extensions and/or future editions of this document will define such situations.
- 7) All members of $EDGES$ and all members of $NODES$ are deleted from CG .

NOTE 196 — The effect of this is that all graph elements referenced by the results of evaluating the <delete statement>, for all records in the current working table, are deleted atomically. If any such graph element cannot be deleted, then no other graph element will be deleted.
- 8) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GD03, “DELETE statement: subquery support”, conforming GQL language shall not contain a <delete item> that contains a <procedure body>.
- 2) Without Feature GD04, “DELETE statement: simple expression support”, conforming GQL language shall not contain a <delete item> that simply contains a <value expression> that is not a <binding variable reference>.

13.6 <call data-modifying procedure statement>

Function

Execute a data-modifying procedure.

Format

<call data-modifying procedure statement> ::=
 <call procedure statement>

Syntax Rules

- 1) Let *CDPS* be the <call data-modifying procedure statement>, let *CPS* be the <call procedure statement> immediately contained in *CDPS*, and let *PC* be the <procedure call> immediately contained in *CPS*.
- 2) If *PC* is an <inline procedure call> that immediately contains the <nested procedure specification> *PROC*, then *PROC* shall immediately contain a <data-modifying procedure specification>.
- 3) If *PC* is a <named procedure call> that immediately contains the <procedure reference> that identifies a procedure *PROC*, then *PROC* shall have the DATA PROCEDURE indication.

General Rules

None.

Conformance Rules

None.

14 Query statements

14.1 <composite query statement>

Function

Set the current working table to the result of a <composite query expression>.

Format

```
<composite query statement> ::=
  <composite query expression>
```

Syntax Rules

- 1) Let *CQS* be the <composite query statement> and let *CQE* be the <composite query expression> immediately contained in *CQS*.
- 2) The declared type of the outgoing working record of *CQS* is the declared type of the incoming working record of *CQS*.
- 3) The declared type of the outgoing working table of *CQS* is the declared type of *CQE*.
- 4) The declared type of *CQS* is the declared type of *CQE*.

General Rules

- 1) Let *NEW_TABLE* be the result of *CQE*.
- 2) The current working table is set to *NEW_TABLE*.
- 3) The current execution outcome is set to a successful outcome with *NEW_TABLE* as its result.

Conformance Rules

None.

14.2 <composite query expression>

Function

Specify binding table compositions.

Format

```
<composite query expression> ::=  
    <composite query expression> <query conjunction> <composite query primary>  
    | <composite query primary>  
  
<query conjunction> ::=  
    <set operator>  
    | OTHERWISE  
  
<set operator> ::=  
    UNION [ <set quantifier> ]  
    | EXCEPT [ <set quantifier> ]  
    | INTERSECT [ <set quantifier> ]  
  
<composite query primary> ::=  
    <linear query statement>
```

Syntax Rules

- 1) If <set operator> is specified and <set quantifier> is not specified, then DISTINCT is implicit.
- 2) Let *CQE* be the <composite query expression>.
- 3) If a <query conjunction> *QC* is immediately contained in *CQE*, then every <query conjunction> directly contained in *CQE* shall be *QC*.
- 4) If *CQE* directly contains a <focused linear query statement>, then *CQE* shall not directly contain an <ambient linear query statement>.

NOTE 197 — As a consequence of this rule, focused statements and ambient statement are mutually exclusive within a <composite query expression>, so that the following is true as well: If *CQE* directly contains an <ambient linear query statement>, then *CQE* does not directly contain a <focused linear query statement>.

- 5) If *CQE* directly contains a <primitive result statement> that is FINISH, then *CQE* shall directly contain at most one <composite query primary>.
- 6) Let *CQP* be the <composite query primary> immediately contained in *CQE*.
- 7) Let *LQS* be the <linear query statement> immediately contained in *CQP*.
- 8) The declared type of the incoming working record of *LQS* is the declared type of the incoming working record of *CQE*.
- 9) The declared type of the incoming working table of *LQS* is the declared type of the incoming working table of *CQE*.
- 10) Case:
 - a) If <query conjunction> is specified, then:
 - i) Let *ICQE* be the <composite query expression> immediately contained in *CQE*.

ISO/IEC 39075:2024(en)
14.2 <composite query expression>

- ii) The declared type of the incoming working record of *ICQE* is the declared type of the incoming working record of *CQE*.
- iii) The declared type of the incoming working table of *ICQE* is the declared type of the incoming working table of *CQE*.
- iv) Let *FCQE* be the set of the columns of the declared type of the outgoing working table of *ICQE* and let *FLQE* be the set of the columns of the declared type of the outgoing working table of *CQP*.
- v) *FCQE* and *FLQE* shall be column name-equal and column-combinable.
- vi) Let *COLS* be the combined columns of *FCQE* and *FLQE*.
- vii) If <set operator> is UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT, then:
 - 1) There shall be no column *COL* in *COLS* such that all of the following is true:
 - A) The value type of *COL* is a dynamic union type *DUT*.
 - B) There exists a pair of different component types *CT1* and *CT2* of *DUT* such that *CT1* and *CT2* are not comparable value types.
 - 2) Each column in *COLS* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 22.15, “Grouping operations”, apply.
- viii) The declared type of *CQE* is the material binding table whose set of columns is *COLS*.
- b) Otherwise, the declared type of *CQE* is the declared type of the outgoing working table of *CQP*.

General Rules

- 1) The result of *CQP* is the result of *LQS*.
- 2) Case:
 - a) If <query conjunction> is specified, then:
 - i) Let *ICQER* be the binding table that is the result of *ICQE*.
 - ii) Let *CQPR* be the binding table that is the result of *CQP*.
 - iii) Let *FCQER* be a binding table determined as follows.
 - Case:
 - 1) If UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT is specified, and there exists a record *R1* of *ICQER* and a record *R2* of *CQPR* such that *R1* and *R2* are not comparable values, then an exception condition is raised: *data exception — values not comparable (22G04)*.
 - 2) If <set operator> is specified, then *FCQER* is new binding table whose columns are *COLS* and whose records are determined by considering each record *R* that is a duplicate of some record in *ICQER* or of some record in *CQPR* or both:
 - A) Let *M* be the number of duplicates of *R* in *ICQER* and let *N* be the number of duplicates of *R* in *CQPR*, where $M \geq 0$ (zero) and $N \geq 0$ (zero).
 - B) If DISTINCT is specified or implicit, then
- Case:

ISO/IEC 39075:2024(en)
14.2 <composite query expression>

- I) If UNION is specified, then *FCQER* contains exactly one duplicate of *R*.
NOTE 198 — *R* exists as a result of [General Rule 2\)a\)iii\)2\)](#), guaranteeing that *FCQER* always contains a duplicate of *R*.
- II) If EXCEPT is specified, then
Case:
1) If $M > 0$ (zero) and $N = 0$ (zero), then *FCQER* contains exactly one duplicate of *R*.
2) Otherwise, *FCQER* contains no duplicate of *R*.
- III) If INTERSECT is specified, then
Case:
1) If $M > 0$ (zero) and $N > 0$ (zero), then *FCQER* contains exactly one duplicate of *R*.
2) Otherwise, *FCQER* contains no duplicates of *R*.
- C) If ALL is specified, then
Case:
I) If UNION is specified, then the number of duplicates of *R* that *FCQER* contains is $(M + N)$.
II) If EXCEPT is specified, then the number of duplicates of *R* that *FCQER* contains is the maximum of $(M - N)$ and 0 (zero).
III) If INTERSECT is specified, then the number of duplicates of *R* that *FCQER* contains is the minimum of M and N .
- 3) If OTHERWISE is specified, then
Case:
A) If *ICQER* contains at least one record, then *FCQER* is *ICQER*.
B) Otherwise, *FCQER* is *CQPR*.
- iv) The result of *CQE* is *FCQER*.
- b) Otherwise, the result of *CQE* is the result of *CQP*.

Conformance Rules

- 1) Without Feature GQ02, “Composite query: OTHERWISE”, conforming GQL language shall not contain a <composite query expression> that immediately contains a <query conjunction> OTHERWISE.
- 2) Without Feature GQ03, “Composite query: UNION”, conforming GQL language shall not contain a <composite query expression> that immediately contains a <query conjunction> UNION.
- 3) Without Feature GQ04, “Composite query: EXCEPT DISTINCT”, conforming GQL language shall not contain a <composite query expression> that immediately contains a <query conjunction> EXCEPT.
- 4) Without Feature GQ05, “Composite query: EXCEPT ALL”, conforming GQL language shall not contain a <composite query expression> that immediately contains a <query conjunction> EXCEPT ALL.

ISO/IEC 39075:2024(en)
14.2 <composite query expression>

- 5) Without Feature GQ06, “Composite query: INTERSECT DISTINCT”, conforming GQL language shall not contain a <composite query expression> that immediately contains a <query conjunction> INTERSECT.
- 6) Without Feature GQ07, “Composite query: INTERSECT ALL”, conforming GQL language shall not contain a <composite query expression> that immediately contains a <query conjunction> INTERSECT ALL.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

14.3 <linear query statement> and <simple query statement>

Function

Specify a linear composition of <simple query statement>s that returns a result.

Format

```
<linear query statement> ::=
  <focused linear query statement>
  | <ambient linear query statement>

<focused linear query statement> ::=
  [ <focused linear query statement part>... ]
  <focused linear query and primitive result statement part>
  | <focused primitive result statement>
  | <focused nested query specification>
  | <select statement>

<focused linear query statement part> ::=
  <use graph clause> <simple linear query statement>

<focused linear query and primitive result statement part> ::=
  <use graph clause> <simple linear query statement> <primitive result statement>

<focused primitive result statement> ::=
  <use graph clause> <primitive result statement>

<focused nested query specification> ::=
  <use graph clause> <nested query specification>

<ambient linear query statement> ::=
  [ <simple linear query statement> ] <primitive result statement>
  | <nested query specification>

<simple linear query statement> ::=
  <simple query statement>...

<simple query statement> ::=
  <primitive query statement>
  | <call query statement>

<primitive query statement> ::=
  <match statement>
  | <let statement>
  | <for statement>
  | <filter statement>
  | <order by and page statement>
```

Syntax Rules

- 1) Let *LQS* be the <linear query statement>.
- 2) Let *STMSEQ* be the sequence of <simple query statement>s, the <primitive result statement>, the <nested query specification>, and the <select statement> directly contained in *LQS*. Let *N* be the number of elements of *STMSEQ*. For *i*, 1 (one) $\leq i \leq N$, let *STM_i* be the *i*-th element of *STMSEQ*.
- 3) The declared type of the incoming working record of *STM₁* is the declared type of the incoming working record of *LQS*.

ISO/IEC 39075:2024(en)
14.3 <linear query statement> and <simple query statement>

- 4) The declared type of the incoming working table of STM_1 is the declared type of the incoming working table of LQS .
- 5) For $2 \leq i \leq N$:
 - a) The declared type of the incoming working record of STM_i is the declared type of the outgoing working record of STM_{i-1} .
 - b) The declared type of the incoming working table of STM_i is the declared type of the outgoing working table of STM_{i-1} .
- 6) The declared type of LQS is the declared type of STM_N .

General Rules

None.

Conformance Rules

- 1) Without Feature GQ01, "USE graph clause", conforming GQL language shall not contain a <focused linear query statement>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

14.4 <match statement>

Function

Expand the current working table with matches from a graph pattern.

Format

```
<match statement> ::=  
    <simple match statement>  
    | <optional match statement>  
  
<simple match statement> ::=  
    MATCH <graph pattern binding table>  
  
<optional match statement> ::=  
    OPTIONAL <optional operand>  
  
<optional operand> ::=  
    <simple match statement>  
    | <left brace> <match statement block> <right brace>  
    | <left paren> <match statement block> <right paren>  
  
<match statement block> ::=  
    <match statement>...
```

Syntax Rules

- 1) Let *MS* be the <match statement>.
- 2) Let *IREDT* be the declared type of the incoming working record of *MS* and let *IREDTFNS* be the set of field type names of *IREDT*.
- 3) Let *ITART* be the record type of the declared type of the incoming working table of *MS* and let *ITARTFNS* be the set of field type names of *ITART*.
- 4) If *MS* immediately contains <optional match statement>, then:
 - a) Let *IFNS* be the union of *IREDTFNS* and *ITARTFNS*.
 - b) Let *IBVRL* be a comma-separated list of all incoming <binding variable reference>s corresponding to <binding variable>s whose names are in *IFNS*.
 - c) A <graph pattern binding table> *T* supplies output bindings to a <match statement> *S* if at least one of the following is true:
 - i) *S* is a <simple match statement> that immediately contains *T*.
 - ii) *S* is an <optional match statement> whose <optional operand> is a <simple match statement> that immediately contains *T*.
 - iii) *S* is an <optional match statement> whose <optional operand> immediately contains a <match statement block> *B* and *T* supplies output bindings to a <match statement> immediately contained in *B*.

NOTE 199 — This is a recursive definition. An <exists predicate> contained in *S* provides an example of a <graph pattern binding table> that does not supply output bindings to *S*.

- d) Let *OBVRL* be a comma-separated list of <binding variable reference>s corresponding to <binding variable>s whose names are column names of the declared type of any <graph pattern binding table> that supplies output bindings to *MS* that are not included in *IFNS*.
- e) Let *RETURN* be defined as follows.

Case:

- i) If *OBVRL* is the empty list, then *RETURN* is:

RETURN NO BINDINGS

- ii) Otherwise, *OBVRL* is non-empty and *RETURN* is:

RETURN *OBVRL*

- f) Let *OO* be the <optional operand> simply contained in *MS*.

Case:

- i) If *OO* immediately contains a <simple match statement> *SMS*, then *MS* is effectively replaced by:

```
OPTIONAL CALL (IBVRL) {  
  SMS  
  RETURN  
}
```

- ii) If *OO* immediately contains a <match statement block> *MSB*, then *MS* is effectively replaced by:

```
OPTIONAL CALL (IBVRL) {  
  MSB  
  RETURN  
}
```

NOTE 200 — This transformation is applied recursively if *MSB* contains one or more nested <optional match statement>s. *IBVRL* and *OBVRL* are computed separately for each transformation.

- 5) If *MS* is a <simple match statement>, then:

- a) Let *GPBT* be the <graph pattern binding table> simply contained in *MS*.
- b) Let *GPBTRT* be the record type of the declared type of *GPBT*.
- c) The declared type of the outgoing working record of *MS* is *IREDT* amended with the record type that is *GPBTRT* restricted to the fields identified by *IREDTFNS*.

NOTE 201 — It is possible that the declared type of a graph pattern variable changes by changing the corresponding declared type of the incoming working record or the corresponding declared type of the incoming working table at certain sites.

- d) The declared type of the outgoing working table of *MS* is the binding table type whose record type is *ITART* amended with the record type that is *GPBTRT* without the fields identified by *IREDTFNS*.

NOTE 202 — It is possible that the declared type of a graph pattern variable changes by changing the corresponding declared type of the incoming working record or the corresponding declared type of the incoming working table at certain sites.

- 6) The declared type of *MS* is the empty type.

General Rules

- 1) Let *NEW_TABLE* be a copy of the result of *GPBT* without the columns identified by *IREDTFNS*.

NOTE 203 — After the application of all Syntax Rules, *MS* is a <simple match statement> and thus *GPBT* is available here as defined by [Syntax Rule 5\)a](#)).

- 2) The current working table is set to *NEW_TABLE*.
- 3) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GQ21, “OPTIONAL: Multiple MATCH statements”, conforming GQL language shall not contain an <optional match statement> that contains a <match statement block>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

14.5 <call query statement>

Function

Execute a query.

Format

<call query statement> ::=
 <call procedure statement>

Syntax Rules

- 1) Let *CQS* be the <call query statement>, let *CPS* be the <call procedure statement> immediately contained in *CQS*, and let *PC* be the <procedure call> immediately contained in *CPS*.
- 2) If *PC* is an <inline procedure call> that immediately contains the <nested procedure specification> *PROC*, then *PROC* shall immediately contain a <query specification>.
- 3) If *PC* is a <named procedure call> that immediately contains the <procedure reference> that identifies a procedure *PROC*, then *PROC* shall have the QUERY PROCEDURE indication.

General Rules

None.

Conformance Rules

None.

14.6 <filter statement>

Function

Select a subset of the records of the current working table.

Format

```
<filter statement> ::=  
  FILTER { <where clause> | <search condition> }
```

Syntax Rules

- 1) Let *FS* be the <filter statement>.
- 2) If *FS* immediately contains the <search condition> *SC*, then it is effectively replaced by the <filter statement>:

 FILTER WHERE *SC*
- 3) Let *WC* be the <where clause> immediately contained in *FS*.
- 4) Let *DTIWR* be the declared type of the incoming working record of *FS*.
- 5) Let *DTIWT* be the declared type of the incoming working table of *FS*.
- 6) The declared type of the incoming working record of *WC* is *DTIWR*.
- 7) The declared type of the incoming working table of *WC* is *DTIWT*.
- 8) The declared type of the outgoing working record of *FS* is *DTIWR*.
- 9) The declared type of the outgoing working table of *FS* is *DTIWT*.
- 10) The declared type of *FS* is the empty type.

General Rules

- 1) The current working table is set to the result of *WC*.
- 2) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GQ08, “FILTER statement”, conforming GQL language shall not contain a <filter statement>.

14.7 <let statement>

Function

Add columns to the current working table.

Format

```
<let statement> ::=
  LET <let variable definition list>

<let variable definition list> ::=
  <let variable definition> [ { <comma> <let variable definition> }... ]

<let variable definition> ::=
  <value variable definition>
| <binding variable> <equals operator> <value expression>
```

Syntax Rules

- 1) Let *LS* be the <let statement>.
- 2) Let *LVDL* be the <let variable definition list> immediately contained in *LS*.
- 3) For every <let variable definition> *LVD* immediately contained in *LVDL* and that immediately contains a <binding variable>, *LVD* is effectively replaced by:

VALUE *LVD*
- 4) Let *IREDT* be the declared type of the incoming working record of *LS* and let *IREDTFNS* be the set of field type names of *IREDT*.
- 5) Let *ITART* be the record type of the declared type of the incoming working table of *LS* and let *ITARTFNS* be the set of field type names of *ITART*.
- 6) Let *IFNS* be the union of *IREDTFNS* and *ITARTFNS*.
- 7) The name of every <binding variable reference> contained in *LS* shall be in *IFNS*.
- 8) Let *IBVRL* be a comma-separated list of all incoming <binding variable reference>s corresponding to <binding variable>s whose names are in *IFNS*.
- 9) Let the <binding variable definition block> *BVDBLK* be the space-separated concatenation of all <let variable definition>s immediately contained in *LVDL* in the order of their occurrence in *LVDL*.
- 10) Let the <return item list> *RIL* be the comma-separated list of all <binding variable>s simply contained in *LVDL*.
- 11) *LS* is effectively replaced by:

```
CALL ( IBVRL ) {
  BVDBLK
  RETURN RIL
}
```

General Rules

None.

Conformance Rules

- 1) Without Feature GQ09, “LET statement”, conforming GQL language shall not contain a <let statement>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

14.8 <for statement>

Function

Unnest a list or a binding table by expanding the current working table.

Format

```
<for statement> ::=  
  FOR <for item> [ <for ordinality or offset> ]  
  
<for item> ::=  
  <for item alias> <for item source>  
  
<for item alias> ::=  
  <binding variable> IN  
  
<for item source> ::=  
  <list value expression>  
  | <binding table reference value expression>  
  
<for ordinality or offset> ::=  
  WITH { ORDINALITY | OFFSET } <binding variable>
```

Syntax Rules

- 1) Let *FS* be the <for statement>.
- 2) Let *FI* be the <for item> immediately contained in *FS*.
- 3) Let *FIA* be the <for item alias> immediately contained in *FI*.
- 4) Let *ANAME* be the name of the <binding variable> immediately contained in *FIA*.
- 5) Let *FIS* be the <for item source> immediately contained in *FI*.
- 6) If <for ordinality or offset> is specified, then let *FOO* be the <for ordinality or offset>; otherwise, let *FOO* be the zero-length character string.
- 7) The declared type of the incoming working record of *FIS* is the declared type of the incoming working record of *FS* amended with the record type of the declared type of the incoming working table of *FS*.
- 8) The declared type of the incoming working table of *FIS* is the material unit binding table type.
- 9) The declared type of the outgoing working record of *FS* is the declared type of the incoming working record of *FS*.
- 10) Let the character string *PNAME* and the closed record type *NEW_FIELDS_RT* be defined as follows.
 - a) Let *SDT* be defined as follows. If *FIS* immediately contains a <list value expression> *LVE*, then *SDT* is the list element type of the declared type of *LVE*; otherwise, *FIS* immediately contains <binding table reference value expression> *BTRVE* and *SDT* is the record type of the declared type of *BTRVE*.
 - b) Let *EFT* be the field type whose name is *ANAME* and whose value type is *SDT*.
 - c) Case:

- i) If *FOO* is not the zero-length character string, then:
 - 1) *PNAME* is the name of the <binding variable> immediately contained in *FOO*.
 - 2) *PNAME* shall not be equal to the name of *EFT*.
 - 3) Let the value type *PVT* be defined as follows.
Case:
 - A) If *FOO* immediately contains WITH ORDINALITY, then *PVT* is the implementation-defined (ID057) exact numeric type with scale 0 (zero) of list element ordinal positions.
 - B) Otherwise, *FOO* immediately contains WITH OFFSET and *PVT* is the implementation-defined (ID058) exact numeric type with scale 0 (zero) of list element position offsets.
 - 4) Let *PFT* be the field type whose name is *PNAME* and whose value type is *PVT*.
 - 5) The field types of *NEW_FIELDS_RT* comprise *EFT* and *PFT*.
 - ii) Otherwise, *PNAME* is the zero-length character string and the field types of *NEW_FIELDS_RT* comprise *EFT*.
- 11) The declared type of the incoming working record of *SdT* and *NEW_FIELDS_RT* shall be field name-disjoint.
 - 12) The declared type of the outgoing working table of *FS* is the binding table type whose record type is the record type of the declared type of the incoming working table of *FS* amended with *NEW_FIELDS_RT*.
 - 13) The declared type of *FS* is the empty type.

General Rules

- 1) Let *NEW_TABLE* be a new binding table whose type is the declared type of the outgoing working table of *FS*.
- 2) For each record *R* of the current working table in a new child execution context amended with *R* whose working table is the new empty binding table whose columns are the field types of *NEW_FIELDS_RT*:
 - a) Let *FISR* be the result of *FIS*.
 - b) Let the sequence of values *VALS* and the non-negative integer *NVALS* be defined as follows.
Case:
 - i) If *FISR* is a list value *LV*, then *VALS* is *LV* and *NVALS* is the cardinality of *LV*.
 - ii) If *FISR* is a binding table reference value whose referent is *BT*, then *VALS* comprises the records of *BT* in the order determined by *BT* and *NVALS* is the cardinality of *BT*.
NOTE 204 — See Subclause 4.13.3, “Binding table types”.
 - iii) Otherwise, *FISR* is the null value, *VALS* is the empty list values, and *NVALS* is 0 (zero).
 - c) For *j*, $1 \text{ (one)} \leq j \leq NVALS$:
 - i) Let *VR_j* be the record comprising a single field whose name is *ANAME* and whose value is the *j*-th element of *VALS*.

ii) Let VRO_j be defined as follows.

Case:

1) If the <for ordinality or offset> is specified, then:

- A) If FOO immediately contains WITH ORDINALITY, then VRO_j is the record obtained by adding a field to VR_j whose name is $PNAME$ and whose value is j .
- B) If FOO immediately contains WITH OFFSET, then VRO_j is the record obtained by adding a field to VR_j whose name is $PNAME$ and whose value is $j-1$.

2) Otherwise, VRO_j is VR_j .

iii) VRO_j is appended to the current working table.

d) The Cartesian product of R and the current working table is appended to NEW_TABLE .

3) The current working table is set to NEW_TABLE .

4) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GQ10, "FOR statement: list value support", conforming GQL language shall not contain a <for statement> that simply contains a <list value expression>.
- 2) Without Feature GQ23, "FOR statement: binding table support", conforming GQL language shall not contain a <for statement> that simply contains a <binding table reference value expression>.
- 3) Without Feature GQ11, "FOR statement: WITH ORDINALITY", conforming GQL language shall not contain a <for statement> that simply contains a <for ordinality or offset> that is WITH ORDINALITY.
- 4) Without Feature GQ24, "FOR statement: WITH OFFSET", conforming GQL language shall not contain a <for statement> that simply contains a <for ordinality or offset> that is WITH OFFSET.

14.9 <order by and page statement>

Function

Specify, for the current working table, either: the ordering of the records, the number of records to be discarded from the beginning of the table, or the maximum number of records to be retained; or any combination of these.

Format

```
<order by and page statement> ::=  
  <order by clause> [ <offset clause> ] [ <limit clause> ]  
  | <offset clause> [ <limit clause> ]  
  | <limit clause>
```

Syntax Rules

- 1) Let *OPS* be the <order by and page statement>.
- 2) Let *DTIWR* be the declared type of the incoming working record of *OPS*.
- 3) Let *DTIWT* be the declared type of the incoming working table of *OPS*.
- 4) The declared type of the incoming working record of any <order by clause>, <offset clause>, and <limit clause> immediately contained in *OPS* is *DTIWR*.
- 5) The declared type of the incoming working table of any <order by clause>, <offset clause>, and <limit clause> immediately contained in *OPS* is *DTIWT*.
- 6) The declared type of *OPS* is the empty type.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *ORDERED* be the binding table defined as follows.
Case:
 - a) If *OPS* immediately contains the <order by clause> *OBC*, then let *ORDERED* be the result of *OBC*.
 - b) Otherwise, let *ORDERED* be *TABLE*.
- 3) Let *OFFSETED* be the binding table defined as follows.
Case:
 - a) If *OPS* immediately contains the <offset clause> *OC*, then let *OFFSETED* be the result of *OC*.
 - b) Otherwise, let *OFFSETED* be *ORDERED*.
- 4) Let *LIMITED* be the binding table defined as follows.
Case:
 - a) If *OPS* immediately contains the <limit clause> *LC*, then let *LIMITED* be the result of *LC*.

- b) Otherwise, let *LIMITED* be *OFFSETED*.
- 5) The current working table is set to *LIMITED*.
- 6) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GQ12, “ORDER BY and page statement: OFFSET clause”, in conforming GQL language, an <order by and page statement> shall not contain an <offset clause>.
- 2) Without Feature GQ13, “ORDER BY and page statement: LIMIT clause”, in conforming GQL language, an <order by and page statement> shall not contain a <limit clause>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

14.10 <primitive result statement>

Function

Define what to include in a query result.

Format

```
<primitive result statement> ::=  
  <return statement> [ <order by and page statement> ]  
  | FINISH
```

Syntax Rules

- 1) Let *PRS* be the <primitive result statement>.
- 2) Let *DTIWR* be the declared type of the incoming working record of *PRS*.
- 3) Let *DTIWT* be the declared type of the incoming working table of *PRS*.
- 4) If the <return statement> *RS* is specified, then:
 - a) The declared type of the incoming working record of *RS* is *DTIWR*.
 - b) The declared type of the incoming working table of *RS* is *DTIWT*.
 - c) Case:
 - i) If the <order by and page statement> *OPS* is specified, then:
 - 1) The declared type of the incoming working record of *OPS* is *DTIWR*.
 - 2) Case:
 - A) If *OPS* immediately contains an <order by clause> *OBC*, then:
 - I) No <sort key> shall contain a <nested query specification>.
 - II) Let *RETURN_IDENTIFIERS* be the set of all <identifier>s immediately contained in an explicit or implicit <return item alias> contained in *RS*.
 - III) Let *ORDER_REFS* be the set of <binding variable reference>s defined as follows:
 - 1) If *RS* does not contain a <group by clause>, a <set quantifier> *DISTINCT*, or any <return item> containing an <aggregate function>, then *ORDER_REFS* is the union of *RETURN_IDENTIFIERS* and the set of <identifier>s representing all column names of *DTIWT* and field type names of *DTIWR*.
 - 2) If *RS* contains a <group by clause> *GBC*, then *ORDER_REFS* is the union of *RETURN_IDENTIFIERS* and the set of all <binding variable reference>s contained in *GBC*.
 - 3) If *RS* does not contain a <group by clause>, but contains a <set quantifier> *DISTINCT* or a <return item> containing an <aggregate function>, then *ORDER_REFS* is *RETURN_IDENTIFIERS*.

ISO/IEC 39075:2024(en)
14.10 <primitive result statement>

- IV) For every <binding variable reference> *BVR* contained in a <sort key> *SK* contained in *OBC*, if *BVR* is contained in *SK* without an intervening instance of <aggregate function>, then *ORDER_REFS* shall contain an identifier that is equivalent to *BVR*.
 - V) If *RS* does not simply contain a <group by clause> or does not simply contain a <return item> containing an <aggregate function> without an intervening instance of <procedure body>, then no <sort key> contained in *OBC* shall contain an <aggregate function>.
 - VI) Let *RIL* be the <return item list> immediately contained in *RS*, let *CRIL* be a copy of *RIL*, and let *OB_COLS* be the empty set.
 - VII) For every <sort key> *SK* contained in *OBC* that contains an <aggregate function>:
 - 1) Let *AGG* be the <value expression> immediately contained in *SK*.
 - 2) Let *CN* be a new system-generated identifier.
 - 3) Append to *CRIL*:
 , *AGG AS CN*
 - 4) *SK* is replaced in *OBC* by:
 CN
 - 5) Let *OB_COL* be a new column whose name is the canonical name form of *CN* and whose type is the declared type of *AGG*.
 - 6) *OB_COL* is added to *OB_COLS*.
 - VIII) For every <binding variable reference> *REF* in *ORDER_REFS* for which it holds that *RETURN_IDENTIFIERS* contains no element that is equivalent to *REF*:
 - 1) Append to *CRIL*:
 , *REF AS REF*
 - 2) Let *OB_COL* be a new column whose name is the canonical name form of *REF* and whose type is the declared type of *REF*.
 - 3) *OB_COL* is added to *OB_COLS*.
 - IX) *RIL* is effectively replaced by *CRIL*. Let *CRS* be *RS* after this replacement.
 - X) The declared type of the incoming working table of *OPS* is the declared type of the outgoing working table of *CRS*.
 - XI) Let *BTT* be the declared type of the outgoing working table of *OPS* without the columns identified by *OB_COLS*.
 - XII) The declared type of *PRS* and the declared type of the outgoing working table of *PRS* is *BTT*.
- B) Otherwise,

- I) The declared type of the incoming working table of *OPS* is the declared type of the outgoing working table of *RS*.
 - II) The declared type of *PRS* is the declared type of the outgoing working table of *OPS*.
 - III) The declared type of the outgoing working table of *PRS* is the declared type of the outgoing working table of *OPS*.
- ii) Otherwise, the declared type of *PRS* is the declared type of the outgoing working table of *RS*.
- 5) If FINISH is specified, then the declared type of *PRS* is the empty type.

General Rules

- 1) If the <return statement> *RS* is specified, then:
- a) The General Rules of *RS* are applied.
 - b) If *PRS* specifies the <order by and page statement> *OPS*, then:
 - i) The General Rules of *OPS* are applied.
 - ii) If *OPS* immediately contains an <order by clause>, then the current working table is set to a copy of the current working table without any of the columns identified by *OB_COLS*.
 - c) The current execution outcome is set to a successful outcome with the current working table as its result.
- 2) If FINISH is specified, then the current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Without Feature GQ14, “Complex expressions in sort keys”, in conforming GQL language, the <value expression> immediately contained in a <sort key> shall be a <binding variable reference>.
- 2) Without Feature GQ16, “Pre-projection aliases in sort keys”, in conforming GQL language, the <value expression> immediately contained in a <sort key> shall not simply contain any <binding variable reference> that is not a <return item alias> in the preceding <return statement>.
- 3) Without Feature GF20, “Aggregate functions in sort keys”, in conforming GQL language, the <value expression> immediately contained in a <sort key> shall not simply contain an <aggregate function>.
- 4) Without Feature GA07, “Ordering by discarded binding variables”, in conforming GQL language, the <order by clause> directly contained in a <primitive result statement> shall not directly contain a <sort key> that directly contains a <binding variable reference> that is not equivalent to the <identifier> immediately contained in a <return item alias> that is directly contained in the <return statement> unless the referenced binding variable of the <binding variable reference> is defined by an intervening BNF non-terminal instance simply contained in the <sort key>.

14.11 <return statement>

Function

Projection and aggregation of the current working table.

Format

```
<return statement> ::=
    RETURN <return statement body>

<return statement body> ::=
    [ <set quantifier> ] { <asterisk> | <return item list> } [ <group by clause> ]
    | NO BINDINGS

<return item list> ::=
    <return item> [ { <comma> <return item> }... ]

<return item> ::=
    <aggregating value expression> [ <return item alias> ]

<return item alias> ::=
    AS <identifier>
```

Syntax Rules

- 1) Let *RS* be the <return statement>.
- 2) Let *RSB* be the <return statement body> immediately contained in *RS*.
- 3) If a <set quantifier> is not immediately contained in *RSB*, then ALL is the implicit <set quantifier> of *RSB*.
- 4) If a <set quantifier> DISTINCT is specified, then each <return item> *RI* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of [Subclause 22.15, "Grouping operations"](#), apply.
- 5) Let *SQ* be the explicit or implicit <set quantifier> of *RSB*.
- 6) If *RSB* immediately contains an <asterisk>, then:
 - a) The declared type of the incoming working table of *RS* shall not be the unit binding table type.
 - b) *RSB* shall not immediately contain a <group by clause>.
 - c) Let *BVSEQ* be the sequence of all column names of the declared type of the incoming working table of *RS* in ascending order. Let *NBVSEQ* be the number of such binding variables. For *i*, 1 (one) $\leq i \leq NBVSEQ$, let *BV_i* be the *i*-th such binding variable in *BVSEQ*.
 - d) For *i*, 1 (one) $\leq i \leq NBVSEQ$, let the new <return item list> *NEWRIL* be a comma-separated list of <return item>s:

BV_i AS *BV_i*
 - e) *RS* is effectively replaced by the <return statement>:

RETURN *SQ* *NEWRIL*
- 7) Let *RIL* be defined as follows.

Case:

- a) If *RSB* is NO BINDINGS, then *SQ* is ALL and *RIL* is the empty sequence.

NOTE 205 — That is, RETURN NO BINDINGS is equivalent to RETURN ALL with an empty <return item list> and no <group by clause>. The Format does not permit an empty <return item list>, hence the need to define these values artificially.

- b) Otherwise, *RIL* is the <return item list> immediately contained in *RSB*.

- 8) For each <return item> *RI* in *RIL*:

Case:

- a) If the <aggregating value expression> immediately contained in *RI* is a <binding variable reference> *RIBV* and *RI* does not immediately contain a <return item alias>, then *RI* is effectively replaced by:

RIBV AS *RIBV*

- b) Otherwise, *RI* shall immediately contain a <return item alias>.

- 9) For a given <return item> *RI*, the *expression* of *RI* is the <aggregating value expression> immediately contained in *RI* and the *alias name* of *RI* is the canonical name form of the <identifier> immediately contained in the explicit or implicit <return item alias> of *RI*.

- 10) Case:

- a) If *RSB* immediately contains a <group by clause> *GBC*:

- i) Let *GRISSET* be the set of grouping <return item>s contained in *RIL* whose alias name is simply contained in *GBC* and let *NGRI* be the number of such <return item>s in *GRISSET*.

- ii) Let *ARISSET* be the set of aggregating <return item>s contained in *RIL* whose alias name is not simply contained in *GBC* and let *NARI* be the number of such <return item>s in *ARISSET*.

- iii) All <return item>s shall be contained in *GRISSET* or *ARISSET*.

- iv) Let *INSET* be the set of all <grouping element>s simply contained in *GBC* that are not equal to an alias name in *GRISSET*. Let *NINSET* be the number of items in *INSET*.

- v) Let *GROUP_COLS* be the set of columns constructed as follows:

- 1) For every grouping <return item> *GRI* in *GRISSET*:

- A) Let *COL* be a new column whose name is the <return item alias> of *GRI* and whose type is the declared type of the <aggregating value expression> immediately contained in *GRI*.

- B) *COL* is included in *GROUP_COLS*.

- 2) For every <grouping element> *IN* in *INSET*:

- A) Let *COL* be a new column whose name is the <identifier> contained in *IN* and whose type is the declared type of the <binding variable reference> contained in *IN*.

- B) *COL* is included in *GROUP_COLS*.

- vi) Let *GR_TABLE_TYPE* be the declared type of a binding table containing all columns of *GROUP_COLS*.

- vii) Let *ARI_TABLE_TYPE* be the declared type of the incoming working table of *RS* without the fields identified by *GROUP_COLS*.
 - viii) The declared type of the incoming working table of all <return item>s in *ARISSET* is *ARI_TABLE_TYPE*.
 - ix) The declared type of the incoming working record of all <return item>s in *ARISSET* is the declared type of the incoming working record of *RS* amended with the record type of *GR_TABLE_TYPE*.
 - b) Otherwise:
 - i) The declared type of the incoming working table of every <return item> is the declared type of the incoming working table of *RS*.
 - ii) The declared type of the incoming working record of every <return item> is the declared type of the incoming working record of *RS*.
- 11) Let *RETURN_TABLE_TYPE* be the declared type of a binding table defined as follows.
- Case:
- a) If *RSB* immediately contains an <asterisk>, then *RETURN_TABLE_TYPE* is the declared type of the incoming working table of *RS*.
 - b) Otherwise:
 - i) Let *RETURN_COLS* be the sequence of columns constructed by, for every element *RI* of *RIL*, creating a column whose name is the alias name of *RI* and whose declared type is the declared type of the expression of *RI*.
 - ii) *RETURN_TABLE_TYPE* is the declared type of a material binding table whose set of columns is *RETURN_COLS*.
- 12) The declared type of the outgoing working table of *RS* is *RETURN_TABLE_TYPE*.
- 13) The declared type of the outgoing working record of *RS* is the declared type of the incoming working record of *RS*.
- 14) The declared type of *RS* is the empty type.

General Rules

- 1) Let *TABLE* be the current working table. Let *N* be the number of records of *TABLE*. For *i*, 1 (one) $\leq i \leq N$, let *R_i* be the *i*-th record of *TABLE* in the order determined by iterating over *TABLE*.
- 2) Let *RETURN_TABLE* be a new empty binding table of type *RETURN_TABLE_TYPE*.
- 3) Case:
 - a) If *RS* immediately contains a <group by clause> *GBC*:
 - i) For *i*, 1 (one) $\leq i \leq N$, the *grouping record GR_i* of a record *R_i* of *TABLE* is a new record constructed as follows:
 - 1) For *k*, 1 (one) $\leq k \leq NGRI$, *GR_i* includes a field *F_k* such that the name of *F_k* is the alias name of *GRISSET_k* and the value of *F_k* is the result of the expression of *GRISSET_k* in a new child execution context amended with *R_i*.

- 2) For j , $1 \text{ (one)} \leq j \leq NINSET$, GR_i includes a field G_j such that the name of G_j is the name specified by the <identifier> contained in $INSET_j$ and the value of G_j is the value in R_i of the <binding variable reference> contained in $INSET$.
- ii) Let GR_TABLE be a new binding table of type GR_TABLE_TYPE of all grouping records GR_i , $1 \text{ (one)} \leq i \leq N$, of all records R_i of $TABLE$.
- iii) Let $GROUP\_BY$ be the result of GBC in a new child execution context with GR_TABLE as its working table.

NOTE 206 — The result of GBC is obtained from the implicit invocation of the General Rules of Subclause 16.15, "<group by clause>".

- iv) For each record K in $GROUP\_BY$:
 - 1) Let $PART$ be a new binding table of type ARI_TABLE_TYPE comprising only the records R_j , $1 \text{ (one)} \leq j \leq N$, from $TABLE$, for which the grouping record GR_j for R_j is not distinct from K , without the columns identified by $GROUP\_COLS$.
 - 2) Let IWR be a copy of the current working record amended with GR_j .
 - 3) Let NR be a copy of K without the fields identified by $INSET$.
 - 4) For every <return item> ARI_j in $ARISSET$, $1 \text{ (one)} \leq j \leq NARI$, append a field to NR whose name is the alias name of ARI_j and whose value is the result of evaluating the <aggregating value expression> of ARI_j in a new child execution context with IWR as its working record and with $PART$ as its working table.

NOTE 207 — This is used to determine the result of <aggregate function>s.

- 5) Append NR to $RETURN\_TABLE$.

b) Otherwise, for each record R of $TABLE$ in a new child execution context amended with R :

- i) Let S be a record defined as follows.

Case:

- 1) If RIL is an empty sequence, then S is the unit record.
- 2) Otherwise, S is the record comprising a field F for every <return item> RI in RIL such that the name of F is the alias name of RI and the value of F is the result of the expression of RI .

- ii) S is added to $RETURN\_TABLE$.

- 4) If SQ is $DISTINCT$, then the current working table is set to duplicate-free copy of $RETURN\_TABLE$ in which every record is distinct from every other record of $RETURN\_TABLE$; otherwise, the current working table is set to $RETURN\_TABLE$.
- 5) Let $FINAL\_TABLE$ be a new binding table of type $RETURN\_TABLE\_TYPE$ obtained from the current working table by determining the preferred column name sequence to be the sequence of alias names of all <return item>s from RIL in the order of their occurrence in RIL .
- 6) The current working table is set to $FINAL\_TABLE$.
- 7) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

- 1) Conforming GQL language shall not contain a <return statement body> that contains NO BINDINGS.

NOTE 208 — A <return statement body> that contains NO BINDINGS is a specification device for use by a syntactic transformation in [Subclause 14.4](#), “<match statement>” to specify the semantics of <optional match statement> if the latter does not supply any new output bindings and is not syntax available to the user.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

14.12 <select statement>

Function

Provide an SQL-style query over graph data, which produces a binding table result.

Format

```
<select statement> ::=
  SELECT [ <set quantifier> ] { <asterisk> | <select item list> }
    [ <select statement body>
      [ <where clause> ]
      [ <group by clause> ]
      [ <having clause> ]
      [ <order by clause> ]
      [ <offset clause> ] [ <limit clause> ] ]

<select item list> ::=
  <select item> [ { <comma> <select item> }... ]

<select item> ::=
  <aggregating value expression> [ <select item alias> ]

<select item alias> ::=
  AS <identifier>

<having clause> ::=
  HAVING <search condition>

<select statement body> ::=
  FROM { <select graph match list> | <select query specification> }

<select graph match list> ::=
  <select graph match> [ { <comma> <select graph match> }... ]

<select graph match> ::=
  <graph expression> <match statement>

<select query specification> ::=
  <nested query specification>
  | <graph expression> <nested query specification>
```

Syntax Rules

- 1) The Syntax Rules of this Subclause are applied before all other Syntax Rules.
- 2) Let *SELECTSTM* be the <select statement>.
- 3) If *SELECTSTM* does not immediately contain a <set quantifier>, then ALL is implicit.
- 4) Let *SETQ* be the explicit or implicit <set quantifier> that is immediately contained in *SELECTSTM*.
- 5) If *SETQ* is DISTINCT, then each <select item> immediately contained in the <select item list> immediately contained in *SELECTSTM* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of [Subclause 22.15](#), “Grouping operations”, apply.
- 6) Let *SSB* be defined as follows. If <select statement body> is specified, then *SSB* is the <select statement body>; otherwise, *SSB* is the zero-length character string.
- 7) If *SELECTSTM* immediately contains an <asterisk>, then:

- a) *SSB* shall not be the zero-length character string.
- b) *SELECTSTM* shall not immediately contain a <group by clause>.
- c) Let *FROM_ITEMS* be the set of names defined as follows.

Case:

- i) If the <select statement body> of *SELECTSTM* immediately contains a <select graph match list> *SGML*, then *FROM_ITEMS* is the set of names of binding variables referenced by all <binding variable reference>s that are declared in <graph pattern>s directly contained in *SGML* without an intervening instance of (another) <graph pattern>.
- ii) Otherwise, the <select statement body> of *SELECTSTM* immediately contains a <select query specification> *SQS* and *FROM_ITEMS* is the set of column names of the declared type of the outgoing working table of the <nested query specification> immediately contained in *SQS*.

- d) Let *FNSEQ* be a sequence of all names of *FROM_ITEMS* and let *NFNSEQ* be the number of such names.
- e) For i , $1 \text{ (one)} \leq i \leq \text{NFNSEQ}$, let *FROM_ITEM_i* be the i -th name in *FNSEQ* and let *BVR_i* be an <identifier> whose canonical name form is *FROM_ITEM_i*.
- f) Let the <select item list> *NEWSIL* be the comma-separated list of <select item>s defined as follows:

BVR₁ AS *BVR₁*, ..., *BVR_{NFNSEQ}* AS *BVR_{NFNSEQ}*

- g) The <asterisk> immediately contained in *SELECTSTM* is effectively replaced by the <select item list>:

NEWSIL

- 8) For each explicit <select item> *SI* immediately contained in the <select item list> immediately contained in *SELECTSTM*,

Case:

- a) If the <aggregating value expression> immediately contained in *SI* is a <binding variable reference> *SIBVR* and *SI* does not immediately contain a <select item alias>, then *SI* is effectively replaced by:

SIBVR AS *SIBVR*

- b) Otherwise, *SI* shall immediately contain a <select item alias>.

- 9) Let *FILTERSTM* be defined as follows.

Case:

- a) If a <where clause> *WHERECL* is specified, then *FILTERSTM* is:

FILTER WHERECL

- b) Otherwise,

Case:

- i) If *SSB* contains the <select graph match list> *SGML* and if the last <select graph match> in *SGML* immediately contains a <match statement> that contains a <graph pattern>

GP that contains a <graph pattern where clause> *GPWC* that is not followed by a <yield clause>, then *FILTERSTM* is:

FILTER GPWC

and *GPWC* is removed from *GP*.

ii) Otherwise, *FILTERSTM* is the zero-length character string.

- 10) Let *XOISIL* be the list of explicit or implicit <select item>s immediately contained in *SELECTSTM* in the order of their appearance, let *XOISISET* be the set of <select item>s in *XOISIL*, let *XOISIASSEQ* be the sequence of <identifier>s immediately contained in <select item alias>es in *XOISIL*, and let *NUM_SIS* be the number of elements of *XOISIL*.
- 11) Let *AGGREGATING_ITEMS* be the set of all <select item>s in *XOISISET* that directly contain an <aggregate function> and let *NUM_AGGREGATING* be the number of elements of *AGGREGATING_ITEMS*.
- 12) Let the <group by clause> *GROUP_BY* and the set of <return item>s *GKRISSET* be determined as follows.

Case:

a) If *SELECTSTM* immediately contains a <group by clause> *GBC* that does not simply contain an <empty grouping set>, then:

i) Let *GEL* be the <grouping element list> immediately contained in *GBC*.

ii) Let *NUM_GE* be the number of elements of *GEL*.

iii) For $j, 1 \text{ (one)} \leq j \leq \text{NUM\_GE}$:

1) Let *GE_j* be the j -th <grouping element> contained in *GEL*.

2) Let *GEID_j* be a new system-generated regular identifier.

3) The following <return item> is added to *GKRISSET*:

GE_j AS GEID_j

iv) *GROUP_BY* is:

GROUP BY GE₁, . . . , GE_{NUM_GE}

b) Otherwise, *GROUP_BY* is the zero-length character string and *GKRISSET* is the empty set.

- 13) Let *NONAGGREGATING_ITEMS* be the set of <select item>s determined as follows:

a) Initially, *NONAGGREGATING_ITEMS* is the empty set.

b) For every <select item> *SI* in *XOISISET* not included in *AGGREGATING_ITEMS*,

Case:

i) If *GKRISSET* is non-empty, then:

1) Let *COSI* be a copy of *SI* with every simply contained <binding variable reference> that is a <value expression> of a <return item> *RI* in *GKRISSET* replaced with the <identifier> immediately contained in the <return item alias> immediately contained in *RI*.

2) *COSI* is added to *NONAGGREGATING_ITEMS*

ii) Otherwise, *SI* is added to *NONAGGREGATING_ITEMS*.

14) Let the set of <return item>s *SARISSET* be determined as follows.

Case:

a) If *SELECTSTM* immediately contains an <order by clause> *OBC*, then:

- i) Initially, *SARISSET* is the empty set.
- ii) For every <sort key> *SK* simply contained in *OBC* that directly contains an <aggregate function>:
 - 1) Let *OBVE* be the <value expression> immediately contained in *SK*.
 - 2) Let *OBVEID* be a new system-generated regular identifier.
 - 3) The following <return item> is added to *SARISSET*:

OBVE AS OBVEID

b) Otherwise, *SARISSET* is the empty set.

15) Let the sets of <return item>s *ARISSET* and *PARISSET* be determined as follows.

Case:

a) If *AGGREGATING_ITEMS* is not the empty set, then:

- i) Initially, both *ARISSET* and *PARISSET* are the empty set.
- ii) For k , $1 \text{ (one)} \leq k \leq \text{NUM\_AGGREGATING\_ITEMS}$:
 - 1) Let *ITEM_k* be the k -th element of *AGGREGATING_ITEMS*.
 - 2) *ITEM_k* shall immediately contain a <select item alias>. Let *ITEM_ALIAS_k* be the <identifier> immediately contained in that <select item alias>.
 - 3) Let *ITEM_EXPR_k* be the <value expression> immediately contained in *ITEM_k*.
 - 4) Let *ITEM_ID_k* be a new system-generated regular identifier.
 - 5) The following <return item> is added to *ARISSET*:

ITEM_EXPR_k AS ITEM_ID_k

6) The following <return item> is added to *PARISSET*:

ITEM_ID_k AS ITEM_ALIAS_k

b) Otherwise, both *ARISSET* and *PARISSET* are the empty set.

16) Let the set of <return item>s *HARISSET* and the <filter statement> *HAVING_FILTER* be defined as follows.

Case:

a) If *SELECTSTM* immediately contains a <having clause> *HC*, then:

- i) Let *HCCOND* be the <search condition> immediately contained in *HC*.
- ii) Let *HCBVE* be the <boolean value expression> immediately contained in *HCCOND*.
- iii) Let *HCID* be a new system-generated regular identifier.
- iv) *HARISSET* is the set comprising the <return item>:

HCBVE AS HCID

v) *HAVING_FILTER* is:

FILTER WHERE HCID

b) Otherwise, *HARISSET* is the empty set and *HAVING_FILTER* is the zero-length character string.

17) Let *LET_ITEMS* be determined as follows.

Case:

a) If *XOISSET* does not directly contain an <aggregate function> and *SELECTSTM* immediately contains an <order by clause> *LOBC* that directly contains a <sort key> that is not a <binding variable reference> and does not contain an <aggregate function>:

i) Initially, *LET_ITEMS* is the empty set.

ii) For every <sort key> *LSK* directly contained in *LOBC* that is not a <binding variable reference> and that does not contain an <aggregate function>:

1) Let *LET_IDENT* be a new system-generated regular identifier.

2) Let *LET_EXPR* be the <value expression> immediately contained in *LSK*.

3) The following <let variable definition> is added to *LET_ITEMS*:

LET_IDENT = *LET_EXPR*

b) Otherwise, *LET_ITEMS* is the empty set.

18) Let *LETSTM* be defined as follows.

Case:

a) If *LET_ITEMS* is non-empty, then:

i) Let *NUM_LET_ITEMS* be the number of elements in *LET_ITEMS*.

ii) For $l, 1 \text{ (one)} \leq l \leq \text{NUM\_LET\_ITEMS}$, let *LET_ITEM_l* be the *l*-th element of *LET_ITEMS*.

iii) *LETSTM* is the <let statement>:

LET LET_ITEM₁, ..., LET_ITEM_{NUM_LET_ITEMS}

b) Otherwise, *LETSTM* is the zero-length character string.

19) If *SELECTSTM* immediately contains a <select item list> that directly contains an <aggregate function> or if *SELECTSTM* immediately contains a <group by clause> that does not simply contain an <empty grouping set>, then for every <sort key> *SESK* directly contained in *SELECTSTM*,

Case:

a) If *SESK* is a <binding variable reference> *SEBVR*, then *SEBVR* shall be equivalent to an <identifier> immediately contained in a <select item alias> directly contained in *SELECTSTM* or a <regular identifier> that is a <grouping element> directly contained in *SELECTSTM*.

b) Otherwise, for every <binding variable reference> *OSEBVR* that is directly contained in *SESK*, *OSEBVR* shall be equivalent to a <grouping element> of a <group by clause> immediately contained in *SELECTSTM*.

20) Let *ORDER_BY* be defined as follows.

Case:

a) If *SELECTSTM* immediately contains an <order by clause> *OIOBC*, then:

i) Let *OISEQ* be the sequence determined as follows.

1) Initially, *OISEQ* is the empty sequence.

2) For every <sort key> *OISK* simply contained in *OIOBC*,

Case:

A) If *SARISSET* is not the empty set and *OISK* directly contains an <aggregate function>, then the <identifier> immediately contained in the <return item alias> immediately contained in a <return item> in *ARISSET* whose <value expression> is *OISK* is added to *OISEQ*.

B) If *GKRISSET* is not the empty set and *OISK* is not a <binding variable reference>, then:

I) Let *COOISK* be a copy of *OISK* in which all simply contained <binding variable reference>s that are the <value expression> of a <return item> *OIRI* in *GKRISSET* are replaced with the <identifier> immediately contained in the <return item alias> immediately contained in *OIRI*.

II) *COOISK* is added to *OISEQ*.

C) If *LET_ITEMS* is not the empty set and *OISK* is a <value expression> immediately contained in a <let variable definition> *OILVD*, then the <binding variable> immediately contained in *OILVD* is added to *OISEQ*.

D) Otherwise, *OISK* is added to *OISEQ*.

ii) Let *NUM_OIS* be the number of elements of *OISEQ*.

iii) For $m, 1 \leq m \leq \text{NUM\_OIS}$, let OI_m be the m -th element of *OISEQ*.

iv) *ORDER_BY* is the comma-separated concatenation of the elements of *OISEQ*:

$OI_1, \dots, OI_{\text{NUM\_OIS}}$

b) Otherwise, *ORDER_BY* is the zero-length character string.

21) Let *RETURN_FIRST* be defined as follows.

Case:

a) If *AGGREGATING_ITEMS* is non-empty, then:

i) Let *FIRST_ITEMS* be the union of *GKRISSET*, *SARISSET*, *ARISSET*, and *HARISSET*.

ii) Let *NUM_FIRST* be the number of elements of *FIRST_ITEMS*.

iii) Let *FSTRIL* be a permutation of *FIRST_ITEMS*. For $p, 1 \leq p \leq \text{NUM\_FIRST}$, let *FIRST_ITEM_p* be the p -th element of *FSTRIL*.

iv) *RETURN_FIRST* is the comma-separated concatenation of the elements of *FIRST_ITEMS*:

$\text{FIRST\_ITEM}_1, \dots, \text{FIRST\_ITEM}_{\text{NUM\_FIRST}}$.

b) Otherwise, *RETURN_FIRST* is the zero-length character string.

22) Let *RETURN_LAST* be defined as follows.

- a) Let *LAST_ITEMS* be the union of *NONAGGREGATING_ITEMS* and *PARISET*.
- b) For $q, 1 \leq q \leq \text{NUM\_SIS}$, let *LAST_ITEM_q* be defined as follows:
 - i) Let *ALIAS* be the q -th element of *XOISIASEQ*.
 - ii) *LAST_ITEM_q* is the element of *LAST_ITEMS* whose <return item alias> immediately contains an <identifier> that is equivalent to *ALIAS*.
- c) *RETURN_LAST* is the comma-separated concatenation of the elements of *LAST_ITEMS*:

$$\text{LAST\_ITEM}_1, \dots, \text{LAST\_ITEM}_{\text{NUM\_SIS}}$$

23) Let *OFFCL* be defined as follows. If <offset clause> is specified, then *OFFCL* is the <offset clause>; otherwise, *OFFCL* is the zero-length character string.

24) Let *LIMCL* be defined as follows. If <limit clause> is specified, then *LIMCL* is the <limit clause>; otherwise, *LIMCL* is the zero-length character string.

25) Let the <variable scope clause> *VSC* and <return item list> *SSBRIL* be defined as follows:

- a) Let *INCVARSEQ* be a sequence of all names of field types and columns of the declared type of the incoming working record and the declared type of the incoming working table and let *NINCVARSEQ* be the number of such names.
- b) For $i, 1 \text{ (one)} \leq i \leq \text{NINCVARSEQ}$, let *INCVAR_i* be the i -th name in *INCVARSEQ* and let *INCBVR_i* be an <identifier> whose canonical name form is *INCVAR_i*.
- c) *VSC* is the comma-separated list of <binding variable reference>s defined as follows:

$$(\text{INCBVR}_1, \dots, \text{INCBVR}_{\text{NINCVARSEQ}})$$
- d) Let *SSBVARSEQ* be the defined as follows.

Case:

- i) If *SSB* immediately contains the <select graph match list> *SGML*, then:
 - 1) Let *SSBVARSET* be the set of all column names of the declared types of the outgoing working tables of all <match statement>s simply contained in *SGML*.
 - 2) *SSBVARSEQ* is a sequence of all names in *SSBVARSET* without the names in *INCVARSEQ*.
- ii) Otherwise, *SSB* immediately contains the <select query specification> *SQS* and *SSBVARSEQ* is the sequence of all column names of the declared type of the <nested query specification> immediately contained in *SQS*.
- e) Let *NSSBVARSEQ* be the number of names in *SSBVARSEQ*.
- f) For $i, 1 \text{ (one)} \leq i \leq \text{NSSBVARSEQ}$, let *SSBVAR_i* be the i -th name in *SSBVARSEQ* and let *SSBBVR_i* be an <identifier> whose canonical name form is *SSBVAR_i*.
- g) *SSBRIL* is the comma-separated list of <return item>s defined as follows:

$$\text{SSBBVR}_1 \text{ AS } \text{SSBBVR}_1, \dots, \text{SSBBVR}_{\text{NSSBVARSEQ}} \text{ AS } \text{SSBBVR}_{\text{NSSBVARSEQ}}$$

26) Let *BODY* be defined as follows.

Case:

- a) If *SSB* immediately contains the <select graph match list> *SGML*, then:
- Let *NUM_SGMS* be the number of <select graph match>es immediately contained in *SGML*.
 - For *r*, $1 \text{ (one)} \leq r \leq \text{NUM\_SGMS}$:
 - Let *SGM_r* be the *r*-th <select graph match> in *SGML*.
 - Let *GREXP_r* be the <graph expression> immediately contained in *SGM_r*.
 - Let *MATCHSTM_r* be the <match statement> immediately contained in *SGM_r*.
 - BODY* is:

```
CALL VSC {  
  USE GREXP1 MATCHSTM1  
  ...  
  USE GREXPNUM_SGMS MATCHSTMNUM_SGMS  
  RETURN SSBRIL  
}
```
- b) If *SSB* immediately contains the <select query specification> *SQS*, then:
- Let *NQS* be the <nested query specification> immediately contained in *SQS*.
 - Case:
 - If *SQS* immediately contains a <graph expression>, then *NQS* shall simply contain an <ambient linear query statement>.
 - Otherwise, *SQS* does not immediately contain a <graph expression> and *NQS* shall simply contain a <focused linear query statement>.
 - Let *USECL* be defined as follows.
Case:
 - If *SQS* immediately contains a <graph expression> *GREXP*, then *USECL* is:

```
USE GREXP
```
 - Otherwise *USECL* is the zero-length character string.
 - BODY* is:

```
CALL VSC {  
  USECL  
  CALL NQS  
  RETURN SSBRIL  
}
```
- c) Otherwise, *BODY* is the zero-length character string.
- 27) After the application of all preceding Syntax Rules,
Case:
- If *BODY* is the zero-length character string, then *SELECTSTM* is equivalent to:

```
RETURN SETQ XOISIL
```


and no further Syntax Rules of this Subclause are applied.

b) Otherwise:

i) Let *REPLACEMENT* be determined as follows:

1) Initially, *REPLACEMENT* is:

BODY
FILTERSTM

2) If *RETURN_FIRST* is not the zero-length character string, then the following is appended to *REPLACEMENT*:

RETURN SETQ RETURN_FIRST
GROUP_BY
NEXT

3) The following is appended to *REPLACEMENT*:

HAVING_FILTER
LETSTM
RETURN SETQ RETURN_LAST
ORDER_BY OFFCL LIMCL

ii) *SELECTSTM* is effectively replaced by *REPLACEMENT*.

General Rules

None.

Conformance Rules

- 1) Without Feature GA07, "Ordering by discarded binding variables", in conforming GQL language, the <order by clause> immediately contained in a <select statement> shall not directly contain a <sort key> that directly contains a <binding variable reference> that is not equivalent to the <identifier> immediately contained in a <select item alias> that is directly contained in the <select statement> unless the referenced binding variable of the <binding variable reference> is defined by an intervening BNF non-terminal instance simply contained in the <sort key>.
- 2) Without Feature GQ14, "Complex expressions in sort keys", in conforming GQL language, the <value expression> immediately contained in a <sort key> shall be a <binding variable reference>.
- 3) Without Feature GQ15, "GROUP BY clause", conforming GQL language shall not contain a <group by clause>.
- 4) Without Feature GQ16, "Pre-projection aliases in sort keys", in conforming GQL language, the <value expression> immediately contained in a <sort key> shall not simply contain a <binding variable reference> that is not a <return item alias> in the preceding <return statement>.

15 Procedure calling

15.1 <call procedure statement> and <procedure call>

Function

Execute a procedure.

Format

```
<call procedure statement> ::=
  [ OPTIONAL ] CALL <procedure call>

<procedure call> ::=
  <inline procedure call>
  | <named procedure call>
```

Syntax Rules

- 1) Let *CPS* be the <call procedure statement>.
- 2) Let *PC* be the <procedure call> immediately contained in *CPS*.
- 3) Let the record type *IREDT* be the declared type of the incoming working record of *CPS*.
- 4) Let *ITART* be the record type of the declared type of the incoming working table of *CPS*.
- 5) Let *IREDTPC* be *IREDT* amended by *ITART*.
- 6) The declared type of the incoming working record of *PC* is *IREDTPC*.
- 7) The declared type of the incoming working table of *PC* is the material unit binding table type.
- 8) Let *OTARTPC* be defined as follows.
Case:
 - a) If *PC* is the <named procedure call> *NPC* and *NPC* has a declared type, then *OTARTPC* is the record type of the declared type of *NPC*.
 - b) If *PC* is the <inline procedure call> *IPC* and *IPC* has a declared type, then *OTARTPC* is the record type of the declared type of *IPC*.
 - c) Otherwise, *OTARTPC* is the material unit record type.
- 9) *ITART* and *OTARTPC* shall be field name-disjoint.
- 10) Let *OTART* be *ITART* amended with *OTARTPC*.
- 11) The declared type of the outgoing working record of *CPS* is *IREDT*.
- 12) The declared type of the outgoing working table of *CPS* is the binding table type whose record type is *OTART*.
- 13) The declared type of *CPS* is the empty type.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *NEW_TABLE* be a new empty binding table whose columns are the field types of *OTART*.
- 3) For each record *R* of *TABLE* in a new child execution context amended with *R*:
 - a) Case:
 - i) If *PC* is an <inline procedure call>, then the General Rules of Subclause 15.2, “<inline procedure call>” are applied.
 - ii) Otherwise, *PC* is a <named procedure call> and the General Rules of Subclause 15.3, “<named procedure call>” are applied.
 - b) Let *RESULT* be a new binding table defined as follows.

Case:

 - i) If the current execution result is a binding table *RESULT_TABLE*, then *RESULT* is *RESULT_TABLE* and the columns of *RESULT* are the field types of *OTARTPC*.
 - ii) Otherwise, the current execution result is omitted and *RESULT* is a unit binding table.
 - c) Let *OPTIONAL_RESULT* be a new binding table defined as follows.

Case:

 - i) If *CPS* immediately contains *OPTIONAL* and *RESULT* is an empty binding table result, then *OPTIONAL_RESULT* comprises a new record of type *OTARTPC* in which every field value is the null value.
 - ii) Otherwise, *OPTIONAL_RESULT* is *RESULT*.
 - d) The Cartesian product of *R* and *OPTIONAL_RESULT* is appended to *NEW_TABLE*.
- 4) The current working table is set to *NEW_TABLE*.
- 5) The current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

None.

15.2 <inline procedure call>

Function

Execute a procedure that is specified inline.

Format

```
<inline procedure call> ::=  
  [ <variable scope clause> ] <nested procedure specification>  
  
<variable scope clause> ::=  
  <left paren> [ <binding variable reference list> ] <right paren>  
  
<binding variable reference list> ::=  
  <binding variable reference> [ { <comma> <binding variable reference> }... ]
```

Syntax Rules

- 1) Let *IPC* be the <inline procedure call>.
- 2) Let *PROC* be the <nested procedure specification> immediately contained in *IPC*.
- 3) If <variable scope clause> is not specified, then:
 - a) Let *FNL* be a comma-separated list of all the field type names of the declared type of the incoming working record of *IPC*.
 - b) *IPC* is effectively replaced by:
$$(FNL) \text{ } PROC$$
- 4) Let *VSC* be the implicit or explicit <variable scope clause> immediately contained in *IPC*.
- 5) For every <binding variable reference> *BVR* simply contained in *VSC*:
 - a) The declared type of the incoming working record of *BVR* is the declared type of the incoming working record of *IPC*.
 - b) The declared type of the incoming working table of *BVR* is the declared type of the incoming working table of *IPC*.
- 6) Let *BVRLN* be defined as follows:
 - a) If *VSC* immediately contains a <binding variable reference list> *BVRL*, then:
 - i) *BVRLN* is the sequence of names of the binding variables referenced by the <binding variable reference>s immediately contained in *BVRL*.
 - ii) *BVRLN* shall not contain two equal names at different positions.
 - b) Otherwise, *BVRLN* is the empty sequence.
- 7) The declared type of the incoming working record of *PROC* is the declared type of the incoming working record of *IPC* comprising the field types identified by *BVRLN*.
- 8) The declared type of the incoming working table of *PROC* is the declared type of the incoming working table of *IPC*.

- 9) The declared type of the outgoing working record of *IPC* is the declared type of the outgoing working record of *PROC*.
- 10) The declared type of the outgoing working table of *IPC* is the declared type of the outgoing working table of *PROC*.
- 11) The declared type of *IPC* is the declared type of *PROC*.

General Rules

- 1) Let *R* be the current working record comprising the fields identified by *BVRLN*.
- 2) The current working record is set to *R*.
- 3) The General Rules of *PROC* are applied.
- 4) The outcome of *IPC* is the outcome of *PROC*.

Conformance Rules

- 1) Without Feature GP01, “Inline procedure”, conforming GQL language shall not contain an <inline procedure call>.
- 2) Without Feature GP02, “Inline procedure with implicit nested variable scope”, in conforming GQL language, an <inline procedure call> shall contain a <variable scope clause>.
- 3) Without Feature GP03, “Inline procedure with explicit nested variable scope”, in conforming GQL language, an <inline procedure call> shall not contain a <variable scope clause>.

15.3 <named procedure call>

Function

Execute a named procedure.

Format

```
<named procedure call> ::=  
  <procedure reference> <left paren> [ <procedure argument list> ] <right paren>  
  [ <yield clause> ]  
  
<procedure argument list> ::=  
  <procedure argument> [ { <comma> <procedure argument> }... ]  
  
<procedure argument> ::=  
  <value expression>
```

Syntax Rules

- 1) Let *NPC* be the <named procedure call> and let *PROC* be the procedure identified by the <procedure reference> immediately contained in *NPC*.
- 2) Let *PROCDESC* be the named procedure descriptor of *PROC*.
- 3) Let *PARAMS* be the list of procedure parameters of *PROCDESC*, let *PARAMSMIN* be the number of procedure parameters required by *PROCDESC* and let *PARAMSMAX* be the maximum number of procedure parameters allowed by *PROCDESC*.
- 4) Let *ARGEXPS* be the sequence of all <value expression>s that are simply contained in *NPC* in the order of their occurrence in *NPC* from left to right and let *NUMARGS* be the number of such elements in *ARGEXPS*.
- 5) *NUMARGS* shall be greater than or equal to *PARAMSMIN*.
- 6) *NUMARGS* shall be less than or equal to *PARAMSMAX*.
- 7) For *i*, $1 \text{ (one)} \leq i \leq \text{NUMARGS}$:
 - a) Let *ARGEXP_i* be the *i*-th element of *ARGEXPS*.
 - b) Let *PT_i* be the declared type of the *i*-th element of *PARAMS*.
 - c) The Syntax Rules of Subclause 22.10, "Store assignment", are applied with a transient site of type *PT_i* as *TARGET* and *ARGEXP_i* as *VALUE*.
- 8) If *NPC* immediately contains a <yield clause>, then the procedure result type of *PROCDESC* shall be a binding table type.
- 9) Let the declared type of *NPC* be defined as follows.

Case:

 - a) If *NPC* immediately contains a <yield clause> *YC*, then the declared type of *NPC* is the declared type of *YC*.
 - b) Otherwise, the declared type of *NPC* is the procedure result type of *PROCDESC*.

General Rules

- 1) For i , $1 \text{ (one)} \leq i \leq \text{PARAMSMAX}$, let ARGVAL_i be defined as follows.
Case:
 - a) If $i \leq \text{NUMARGS}$, then ARGVAL_i is the result of ARGEXP_i .
 - b) Otherwise, ARGVAL_i is the default value of the i -th element of PARAMS .
- 2) For i , $1 \text{ (one)} \leq i \leq \text{PARAMSMAX}$, let R be a new record comprising fields F_i defined as follows:
 - a) Let T_i be the declared type of the i -th element of PARAMS .
 - b) Let TS be a transient site of type T_i . The General Rules of [Subclause 22.10](#), “Store assignment”, are applied with TS as **TARGET** and ARGVAL_i as **VALUE**. Let TSV_i be the value of TS .
 - c) The name of F_i is the procedure parameter name of the i -th element of PARAMS and the value of F_i is TSV_i .
- 3) The following steps are performed in a new child execution context with R as its working record:
 - a) Execute PROC .
 - b) Let RESULT be the result returned from the successful execution of PROC .
 - c) The current execution outcome is set as follows.
Case:
 - i) If a <yield clause> YC is specified, then:
 - 1) The General Rules of YC are applied; let YIELD be the result returned from the application of these General Rules.
 - 2) The current execution outcome is set to a successful outcome with YIELD as its result.
 - ii) Otherwise, set the current execution outcome to a successful outcome with RESULT as its result.

Conformance Rules

- 1) Without Feature GP04, “Named procedure calls”, conforming GQL Language shall not contain a <named procedure call>.
- 2) Without Feature GP15, “Graphs as procedure arguments”, in conforming GQL language, the declared type of a <value expression> immediately contained in a <procedure argument> shall not be a supertype of a graph reference value type.
- 3) Without Feature GP14, “Binding tables as procedure arguments”, in conforming GQL language, the declared type of a <value expression> immediately contained in a <procedure argument> shall not be a supertype of a binding table reference value type.

16 Common elements

16.1 <at schema clause>

Function

Declare a working schema and its scope.

Format

```
<at schema clause> ::=
  AT <schema reference>
```

Syntax Rules

- 1) Let *ASC* be the <at schema clause>.
- 2) Let *ACSR* be an <absolute catalog schema reference> that identifies the GQL-schema identified by the <schema reference> immediately contained in *ASC*.
 NOTE 209 — If the <schema reference> *SR* that is immediately contained in *ASC* is a <relative catalog schema reference> or a <reference parameter specification>, then the Syntax Rules of [Subclause 17.1](#), "<schema reference> and <catalog schema parent and name>" are applied to determine the GQL-schema identified by *SR*.
- 3) Let *PB* be the <procedure body> immediately containing *ASC*.
- 4) The scope clause of *ASC* is *PB*.
- 5) The scope of *ACSR* comprises *PB*.
- 6) *ASC* identifies *ACSR* as a working schema reference.

General Rules

None.

Conformance Rules

None.

16.2 <use graph clause>

Function

Declare a working graph and its scope.

Format

```
<use graph clause> ::=  
  USE <graph expression>
```

Syntax Rules

- 1) Let *UGC* be the <use graph clause>.
- 2) Let *GE* be the <graph expression> simply contained in *UGC*.
- 3) The declared type of the incoming working record of *GE* is the declared type of the incoming working record of *UGC*.
- 4) The declared type of the incoming working table of *GE* is the material unit binding table type.
- 5) Let *PART* be the instance of the BNF non-terminal that immediately contains *UGC*.
- 6) The scope clause of *UGC* is defined as follows.
Case:
 - a) If *PART* is simply contained in a <focused linear query statement> *FLQS*, then the scope clause of *UGC* is *FLQS*.
 - b) Otherwise, *PART* is simply contained in a <focused linear data-modifying statement> *FLDMS*, and the scope clause of *UGC* is *FLDMS*.
- 7) The scope of *GE* is defined as follows.
Case:
 - a) If *PART* is a <focused linear query statement part>, then the scope of *GE* comprises the <simple linear query statement> immediately contained in *PART*.
 - b) If *PART* is a <focused linear query and primitive result statement part>, then the scope of *GE* comprises the <simple linear query statement> and the <primitive result statement> immediately contained in *PART*.
 - c) If *PART* is a <focused primitive result statement>, then the scope of *GE* comprises the <primitive result statement> immediately contained in *PART*.
 - d) If *PART* is a <focused nested query specification>, then the scope of *GE* comprises the <nested query specification> immediately contained in *PART*.
 - e) If *PART* is a <focused linear data-modifying statement body>, then the scope of *GE* comprises the <simple linear query statement>, the <simple data-modifying statement>, the <simple linear data-accessing statement>, and the <primitive result statement> immediately contained in *PART*.

- f) Otherwise, *PART* is a <focused nested data-modifying procedure specification> and the scope of *GE* comprises the <nested data-modifying procedure specification> immediately contained in *PART*.
- 8) *UGC* identifies *GE* as a working graph site.

General Rules

- 1) The working graph site that is *GE* is set to the result of evaluating *GE* in a new child execution context.

Conformance Rules

- 1) Without Feature GQ01, “USE graph clause”, conforming GQL language shall not contain a <use graph clause>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

16.3 <graph pattern binding table>

Function

Evaluate a <graph pattern> to a binding table.

Format

```
<graph pattern binding table> ::=  
  <graph pattern> [ <graph pattern yield clause> ]  
  
<graph pattern yield clause> ::=  
  YIELD <graph pattern yield item list>  
  
<graph pattern yield item list> ::=  
  <graph pattern yield item> [ { <comma> <graph pattern yield item> }... ]  
  | NO BINDINGS  
  
<graph pattern yield item> ::=  
  <element variable reference>  
  | <path variable reference>
```

Syntax Rules

- 1) Let *GPBT* be the <graph pattern binding table> and let *GP* be the <graph pattern> simply contained in *GPBT*.
- 2) Every variable in the scope of *GP* is also in the scope of the <graph pattern yield clause> simply contained in *GPBT*.
- 3) Let *EVSET* be the set of names of element variable declared by *GP*, let *PVSET* be the set of names of path variables declared by *GP*, and let *GPVARS* be a permutation of $EVSET \cup PVSET$ in the order of their first occurrence as an <element variable declaration> or a <path variable declaration>, respectively, that is simply contained in *GP*. Let *SPVSET* be the set of names of subpath variables declared by *GP*.
- 4) Each graph pattern variable name in *GPVARS* shall identify exactly one graph pattern variable in *GP* at the same depth of graph pattern matching.
- 5) Let *GLOBALS* be *GPVARS* restricted to the names of graph pattern variables in the scope of *GP*.
- 6) Let *IREDT* be the declared type of the incoming working record of *GPBT*.
- 7) *IREDT* shall not contain a field whose name is in *SPVSET*.
- 8) Let *ITART* be the record type of the declared type of the incoming working table of *GPBT*.
- 9) *ITART* shall not contain a field whose name is in *SPVSET*.
- 10) Let *RECORD_OVERLAP* be *GLOBALS* restricted to the names of fields of *IREDT*, let *TABLE_OVERLAP* be *GLOBALS* restricted to the names of fields of *ITART*, and let *OVERLAP* be the concatenation of *RECORD_OVERLAP* with *TABLE_OVERLAP*.
- 11) The sequence of variable names *XGPYCVARS* is defined as follows.

Case:

ISO/IEC 39075:2024(en)
16.3 <graph pattern binding table>

- a) If an explicit <graph pattern yield clause> *XGPYC* is specified, then *XGPYCVARS* is the sequence of names of variables identified by instances of <binding variable reference>s simply contained in *XGPYC*.
- b) Otherwise, *XGPYCVARS* is the empty sequence.

12) Case:

- a) If the length of *XGPYCVARS* is at least 1 (one), then:
 - i) *XGPYCVARS* shall not contain two equal names at different positions.
 - ii) Every name in *XGPYCVARS* shall be contained in *GLOBALS*.
 - iii) Let *OGPYCVARS* be *OVERLAP* without the names contained in *XGPYCVARS*.
 - iv) Let *GPYIL* be the comma-separated list of the elements of the concatenation of *OGPYCVARS* with *XGPYCVARS*.
 - v) *XGPYC* is equivalent to:

YIELD *GPYIL*

- b) Otherwise, *XGPYCVARS* is the empty sequence.

Case:

- i) If *GLOBALS* is empty, then the implicit <graph pattern yield clause> of *GPBT* is:

YIELD NO BINDINGS
- ii) Otherwise, *GLOBALS* is non-empty. Let *GPYIL* be the comma-separated list of the elements of *GLOBALS*. The implicit <graph pattern yield clause> of *GPBT* is:

YIELD *GPYIL*

13) Let *GPYC* be the explicit or implicit <graph pattern yield clause> after the application of [Syntax Rule 12](#)).

14) Let *GPYCREFS* be defined as follows.

Case:

- a) If *GPYC* is YIELD NO BINDINGS, then *GPYCREFS* is the empty sequence.
- b) Otherwise, *GPYCREFS* is the sequence of <binding variable reference>s simply contained in *GPYC*.

NOTE 210 — This is restricted to variable names from *GLOBALS*, i.e., in particular contains no names of subpath variables declared by *GP*.

15) The binding graph pattern of each <binding variable reference> in *GPYCREFS* shall be *GP*.

NOTE 211 — See [Syntax Rule 10](#)) of Subclause 20.12, "<binding variable reference>", for the definition of binding graph pattern.

16) For every <binding variable reference> *BVR* with name *NAME* in *GPYCREFS* for which it holds that *NAME* is the name of a field in *IREDT* or *ITART*:

- a) Let *IVT* be the value type of the field type in *IREDT* or *ITART* with name equal to *NAME*.

Case:

ISO/IEC 39075:2024(en)
16.3 <graph pattern binding table>

- i) If the declared type of *BVR* has group degree of reference, then *IVT* shall have group degree of reference.
- ii) Otherwise, the declared type of *BVR* does not have group degree of reference and *IVT* shall not have group degree of reference.

17) Let *GPYCFTSET* be the set of the projected field types of all elements of *GPYCREFS*.

NOTE 212 — See [Syntax Rule 13](#)) of Subclause 20.12, “<binding variable reference>”, for the definition of projected field type.

18) Let *GPYCRT* be the closed record type whose field type set is *GPYCFTSET*.

19) Let *IBRT* be *IREDT* amended with *ITART*.

20) The records bound to *COMBINED* by applications of [General Rule 6\)b](#)) whose declared type is *IBRT* and the binding tables bound to *INNER_TABLE* by applications of [General Rule 6\)a](#)) whose declared type is *GPYCRT* are the operands of an equality operation (the natural join performed by the application of [General Rule 6\)g](#))). The Syntax Rules and Conformance Rules of [Subclause 22.13](#), “Equality operations”, apply.

21) For each <element property specification> *EPS* simply contained in *GP*:

- a) Let *EPP* be the <element pattern predicate> that simply contains *EPS*.
- b) Let *EPF* be the <element pattern filler> that simply contains *EPP*.
- c) Let *EVARDECL* and *EVAR* be defined as follows.

Case:

- i) If *EPF* simply contains an <element variable declaration> *EVARDECL*, then *EVAR* is the <identifier> contained in the <element variable declaration> simply contained in *EPF*.
- ii) Otherwise, *EVAR* is a new system-generated regular identifier distinct from every element variable, subpath variable, and path variable contained in *GP* and *EVARDECL* is TEMP *EVAR*.

NOTE 213 — In this case, *EVAR* specifies a temporary element variable that is never contained in *GPYC*.

- d) Let *EPILE* be defined as follows. If *EPF* simply contains the <is label expression> *EPILE_CAND*, then *EPILE* is *EPILE_CAND*; otherwise, *EPILE* is the zero-length character string.
- e) Let *PECL* be the <property key value pair list> simply contained in *EPS*.
- f) Let *NOPEC* be the number of <property key value pair>s simply contained in *PECL*.
- g) Let *PEC*₁, ..., *PEC*_{NOPEC} be the <property key value pair>s simply contained in *PECL*.
- h) For *i*, 1 (one) ≤ *i* ≤ *NOPEC*:
 - i) Let *PROP*_{*i*} be the <property name> simply contained in *PEC*_{*i*}.
 - ii) Let *VAL*_{*i*} be the <value expression> simply contained in *PEC*_{*i*}.
 - iii) Let *RPEC*_{*i*} be a <comparison predicate> formed as:

$$EVAR.PROP_i = VAL_i$$

- i) Let *EPSC* be a <boolean value expression> formed through the concatenation of <boolean factor>s: *RPEC*₁ AND ... AND ... *RPEC*_{NOPEC}.

ISO/IEC 39075:2024(en)
16.3 <graph pattern binding table>

j) *EPF* is effectively replaced by:

EVARDECL EPILE WHERE EPSC

- 22) Let *GPT* be the <graph pattern> simply contained in *GPBT* after the preceding transformations and let *PPLT* be the <path pattern list> simply contained in *GPT*.
- 23) The current working graph site of *GPBT* shall not be “omitted”.
- 24) Let *CWGS* be the current working graph site of *GPBT*.
- 25) For every <value expression> or <search condition> *EXP* simply contained in *GP*, if *EXP* is evaluated in a new child execution context during the application of the [Subclause 22.6, “Application of bindings to evaluate an expression”](#), then:
- a) The declared type of the incoming working record of *EXP* is the record type whose field types are given by the union between:
- i) *IREDT* without the fields identified by *RECORD_OVERLAP*.
- ii) *ITART* without the fields identified by *TABLE_OVERLAP*.
- iii) The projected field types of all graph pattern variable references simply contained in *EXP* whose variables are declared by *GP* and that are exposed in *EXP*.
- NOTE 214 — See [Syntax Rule 13](#)) of [Subclause 20.12, “<binding variable reference>”](#), for the definition of projected field type.
- b) The declared type of the incoming working table of *EXP* is the unit binding table type.
- 26) Let *GPBTRT* be the closed record type whose field types are given by the union between:
- a) *IREDT* without the fields identified by *RECORD_OVERLAP*.
- b) *ITART* without the fields identified by *TABLE_OVERLAP*.
- c) *GPYCFTSET*.
- 27) The declared type of *GPBT* is the binding table type whose record type is *GPBTRT*.

General Rules

- 1) Let *PG* be the graph referenced by *CWGS*.
- 2) The General Rules of [Subclause 22.2, “Machinery for graph pattern matching”](#), are applied with *PG* as *PROPERTY GRAPH* and *PPLT* as *PATH PATTERN LIST*; let *MACH* be the *MACHINERY* returned from the application of those General Rules.
- 3) Let *MATCH_TABLE* be a new empty binding table whose columns are the columns of the declared type of *GPBT*.
- 4) The preferred column name sequence *PCNS* of *MATCH_TABLE* is the concatenation of the following sequences of names in the order given.

Case:

- a) If the current working table has a preferred column name sequence *PCS*, then *PCNS* is the concatenation of:
- i) The field type names of *IREDT* not included in *XGPYCVARS*, in ascending order.
- ii) *PCS* without the names included in *XGPYCVARS*.

ISO/IEC 39075:2024(en)
16.3 <graph pattern binding table>

- iii) *XGPYCVARS*.
- b) Otherwise, *PCNS* is the concatenation of:
 - i) The field type names of *IREDT* and *ITART* not included in *XGPYCVARS*, in ascending order.
 - ii) *XGPYCVARS*.
- 5) Let *CWR* be the current working record.
- 6) For each record *OUTER* in the current working table:
 - a) Let *INNER_TABLE* be a new empty binding table whose columns are the field types in *GPYCFTSET*.
 - b) Let *COMBINED* be *CWR* amended with *OUTER*. The field types of *COMBINED* are the field types of *IREDT* amended with *ITART*.
 - c) Let *SANITIZED* be *COMBINED* without the fields identified by *OVERLAP*.
 - d) In a new child execution context whose working record is *SANITIZED*, the General Rules of Subclause 16.4, "<graph pattern>", are applied with *PG* as *PROPERTY GRAPH*, *GPT* as *GRAPH PATTERN*, *PPLT* as *PATH PATTERN LIST*, and *MACH* as *MACHINERY*; let *MATCHES* be the *SET OF REDUCED MATCHES* returned from the application of those General Rules.
 - e) For each reduced match *RM* in *MATCHES*:
 - i) The General Rules of Subclause 22.8, "Application of bindings to generate a record", are applied with *GPT* as *GRAPH PATTERN*, *GPYC* as *YIELD CLAUSE*, *RM* as *MULTI-PATH BINDING*, and *MACH* as *MACHINERY*; let *INNER* be the *RECORD* returned from the application of those General Rules.
 - ii) *INNER* is appended to *INNER_TABLE*.
 - f) If there is a record *R1* of *INNER_TABLE* such that *R1* and *COMBINED* are not comparable values, then an exception condition is raised: *data exception — values not comparable* (22G04).
 - g) The binding table constructed as the natural join between *COMBINED* and *INNER_TABLE* is appended to *MATCH_TABLE*.
- 7) The result of *GPBT* is *MATCH_TABLE*.

Conformance Rules

- 1) Conforming GQL language shall not contain a <graph pattern yield item list> that contains NO BINDINGS.

NOTE 215 — A <graph pattern yield item list> that contains NO BINDINGS is a specification device and is not syntax available to the user.
- 2) Without Feature GQ19, "Graph pattern YIELD clause", conforming GQL language shall not contain a <graph pattern binding table> that immediately contains a <graph pattern yield clause>.

16.4 <graph pattern>

Function

Specify a pattern to be matched in a graph.

Subclause Signature

"<graph pattern>" [General Rules] (
Parameter: "PROPERTY GRAPH",
Parameter: "GRAPH PATTERN",
Parameter: "PATH PATTERN LIST",
Parameter: "MACHINERY"
) Returns: "SET OF REDUCED MATCHES"

PROPERTY GRAPH — a property graph

GRAPH PATTERN — a <graph pattern>.

PATH PATTERN LIST — a <path pattern list>.

MACHINERY — the machinery for graph pattern matching

SET OF REDUCED MATCHES — the resulting set of reduced matches.

Format

```
<graph pattern> ::=  
  [ <match mode> ] <path pattern list>  
  [ <keep clause> ]  
  [ <graph pattern where clause> ]  
  
<match mode> ::=  
  <repeatable elements match mode>  
  | <different edges match mode>  
  
<repeatable elements match mode> ::=  
  REPEATABLE <element bindings or elements>  
  
<different edges match mode> ::=  
  DIFFERENT <edge bindings or edges>  
  
<element bindings or elements> ::=  
  ELEMENT [ BINDINGS ]  
  | ELEMENTS  
  
<edge bindings or edges> ::=  
  <edge synonym> [ BINDINGS ]  
  | <edges synonym>  
  
<path pattern list> ::=  
  <path pattern> [ { <comma> <path pattern> }... ]  
  
<path pattern> ::=  
  [ <path variable declaration> ] [ <path pattern prefix> ] <path pattern expression>  
  
<path variable declaration> ::=  
  <path variable> <equals operator>
```


<keep clause> ::=
KEEP <path pattern prefix>

<graph pattern where clause> ::=
WHERE <search condition>

Syntax Rules

- 1) Let *GP* be the <graph pattern>.
- 2) If *BNF1* and *BNF2* are instances of two BNF non-terminals, both contained in *GP* without an intervening <graph pattern>, then *BNF1* and *BNF2* are said to be *at the same depth of graph pattern matching*.
NOTE 216 — *BNF1* can contain *BNF2* while being at the same depth of graph pattern matching.
- 3) In a <path pattern list>, if two <path pattern>s expose an element variable *EV*, then both shall expose *EV* as an unconditional singleton variable.
NOTE 217 — This case expresses an implicit join on *EV*. Implicit joins between conditional singleton variables or group variables are forbidden.
- 4) Two <path pattern>s shall not expose the same subpath variable.
NOTE 218 — Implicit equijoins on subpath variables are not supported.
- 5) The name of a node variable shall not be equivalent to the name of an edge variable declared at the same depth of graph pattern matching.
- 6) If <keep clause> *KP* is specified, then:
 - a) Let *PSP* be the <path pattern prefix> simply contained in *KP*.
 - b) For each <path pattern> *PP* simply contained in *GP*:
 - i) *PP* shall not contain a <path search prefix>.
 - ii) Case:
 - 1) If *PP* specifies a <path variable declaration>, then let *PVDECL* be that <path variable declaration>.
 - 2) Otherwise, let *PVDECL* be the zero-length character string.
 - iii) Case:
 - 1) If *PP* specifies a <path mode prefix>, then let *PMP* be that <path mode prefix>.
 - 2) Otherwise, let *PMP* be the zero-length character string.
 - iv) Let *PPE* be the <path pattern expression> simply contained in *PP*.
 - v) *PP* is effectively replaced by
$$PVDECL \text{ } PSP \text{ } (\text{ } PMP \text{ } PPE \text{ })$$
 - c) The <keep clause> is removed from the <graph pattern>.
- 7) After the preceding transformations, for every <path pattern> *PP*, if *PP* contains a <path pattern prefix> *PPP* that specifies a <path mode> *PM*, then:
 - a) Case:

- i) If *PP* specifies a <path variable declaration>, then let *PVDECL* be that <path variable declaration>.
- ii) Otherwise, let *PVDECL* be the zero-length character string.
- b) Let *PPE* be the <path pattern expression> simply contained in *PP*.
- c) *PP* is effectively replaced by

PVDECL PPP (PM PPE)

NOTE 219 — One effect of the preceding transforms is that every <path mode> expressed outside a <parenthesized path pattern expression> is also expressed within a <parenthesized path pattern expression>. For example,

ALL SHORTEST TRAIL GROUP <path pattern expression>
is rewritten as

ALL SHORTEST TRAIL GROUP (TRAIL <path pattern expression>)

The TRAIL specified outside the parentheses is now redundant. The benefit is that the definition of a consistent path binding in Subclause 22.2, “Machinery for graph pattern matching”, only has to consider <path mode>s declared in <parenthesized path pattern expression>s.

- 8) Let *GPT* be the <graph pattern> after the preceding syntactic transformations.
- 9) Let *PPL* be the <path pattern list> simply contained in *GPT*.
- 10) If *GPT* does not specify a <match mode>, then an implementation-defined (ID086) <match mode> is implicit.
- 11) Let *MM* be the <match mode> implicitly or explicitly specified by *GPT*.
- 12) If *MM* is <different edges match mode> and *PPL* simply contains a <path pattern> that is selective, then *PPL* shall not simply contain any other <path pattern>.

NOTE 220 — If *MM* is <different edges match mode> and there is a selective <path pattern> *SPP*, then *PPL* must only contain *SPP*. If there is no selective <path pattern> in *GPT*, then there are no restrictions on how many non-selective <path pattern>s are contained in *PPL*. If *MM* is <repeatable elements match mode>, then there is no restriction on how many (selective and non-selective) <path pattern>s are contained in *PPL*.

- 13) Let *E* be an element variable declared by *GPT*.

Case:

- a) If *E* is exposed by *GPT* as an unconditional singleton, then *E* is a *global unconditional singleton* of *GPT*.
- b) Otherwise, let *PPPE* be the outermost <parenthesized path pattern expression> that exposes *E* as an unconditional singleton; the *unconditional singleton scope index* of *E* in *GPT* is the bracket index of *PPPE*.

NOTE 221 — Bracket index is defined in Subclause 22.2, “Machinery for graph pattern matching”. The unconditional singleton scope index is well-defined because implicit equijoins between conditional singleton variables or group variables are forbidden. Hence there cannot be two <parenthesized path pattern expression>s that expose *E* as a conditional singleton or group variable unless one is contained in the other. For example,

(((-[E]->) -[E]->) * -[F]->) *

The unconditional singleton scope of *E* is the middle <parenthesized path pattern expression> in the nest of three.

- 14) After the preceding transformations, for every <quantified path primary> *QPP* contained in *GPT*, at least one of the following shall be true:
- The <graph pattern quantifier> of *QPP* is bounded.
 - QPP* is contained in a restrictive <parenthesized path pattern expression>.
 - QPP* is contained in a selective <path pattern>.
 - MM* is <different edges match mode>.
- NOTE 222 — Unless an explicit <path mode> other than WALK is specified, an explicit or implicit specification of a <different edges match mode> effectively imparts the <path mode> TRAIL without the presence of the keyword TRAIL.
- 15) Each <path variable> *PV* contained in *GPT* is the name of a path variable. The *degree of exposure* of the path variable that *PV* identifies is unconditional singleton.

General Rules

- 1) Let *PG* be the *PROPERTY GRAPH*, let *GPT* be the *GRAPH PATTERN*, let *PPL* be the *PATH PATTERN LIST*, and let *MACH* be the *MACHINERY* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *SET OF REDUCED MATCHES*.
- NOTE 223 — In this document, *PG* is always the graph referenced by the current working graph site of *GPT*.
- 2) The following components of *MACH* are identified:
- ABC*, the alphabet, formed as the disjoint union of the following:
 - SVV*, the set of names of node variables.
 - SEV*, the set of names of edge variables.
 - SPS*, the set of subpath symbols.
 - SAS*, the set of anonymous symbols.
 - SBS*, the set of bracket symbols.
 - REDUCE*, the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.
- 3) Let *NP* be the number of <path pattern>s simply contained in *PPL*. Let *PP*₁, ..., *PP*_{*NP*} be the <path pattern>s simply contained in *PPL* after the transformations in the Syntax Rules.
- 4) A multi-path binding *MPBINDING* is *different-edges-matched* if, for every edge binding *EB1* = (*EV1*, *E*) contained in *MPBINDING*, there is no edge binding *EB2* = (*EV2*, *E*) contained in *MPBINDING* at a different position than *EB1* that binds the edge *E*.
- 5) For every *i*, 1 (one) ≤ *i* ≤ *NP*:
- Let *PPE* be the <path pattern expression> simply contained in *PP*_{*i*}.
 - The General Rules of Subclause 22.3, “Evaluation of a <path pattern expression>”, are applied with *PG* as *PROPERTY GRAPH*, *PPL* as *PATH PATTERN LIST*, *MACH* as *MACHINERY*, and *PPE* as *SPECIFIC BNF INSTANCE*; let *SMPPE*_{*i*} be the *SET OF MATCHES* returned from the application of those General Rules.

NOTE 224 — If an elementary variable has been multiply declared within a restrictive <parenthesized path pattern expression> *PP*, then no matches are returned for *PP*. For example:

ISO/IEC 39075:2024(en)
16.4 <graph pattern>

MATCH ACYCLIC (X) -> (X)

does not find any results, even if there are nodes with self-edges.

c) Case:

i) If PP_i is a selective <path pattern>, then:

1) Case:

A) If MM is <different edges match mode>, then let $SMUP_i$ be the set of different-edges-matched multi-path bindings in $SMPPE_i$.

NOTE 225 — If an edge variable has been multiply declared within a <path pattern> PP , then no matches are returned for PP . For example, the following produces no results:

MATCH DIFFERENT EDGES
ANY SHORTEST () -[E]-> () -[E]-> ()

B) Otherwise, let $SMUP_i$ be $SMPPE_i$.

2) The General Rules of Subclause 22.4, “Evaluation of a selective <path pattern>”, are applied with PG as *PROPERTY GRAPH*, PPL as *PATH PATTERN LIST*, $MACH$ as *MACHINERY*, PP_i as *SELECTIVE PATH PATTERN*, and $SMUP_i$ as *INPUT SET OF LOCAL MATCHES*; let SM_i be the *OUTPUT SET OF LOCAL MATCHES* returned from the application of those General Rules.

ii) Otherwise, let SM_i be $SMPPE_i$.

6) Let $CROSS$ be the cross product $SM_1 \times \dots \times SM_{NP}$.

7) Let $INNER$ be the set of multi-path bindings MPB in $CROSS$ such that, for every unconditional singleton <element variable> USV exposed by PPL , USV is bound to a unique graph element by the elementary bindings of USV contained in MPB .

NOTE 226 — Anonymous symbols are not <element variable>s; there is no requirement that two anonymous symbols bind to the same graph element.

8) Case:

a) If MM is <different edges match mode>, then let $BINDINGS$ be the set of different-edges-matched multi-path bindings in $INNER$.

NOTE 227 — If an edge variable has been multiply declared within a <graph pattern> GP , then no matches are returned for GP . For example, the following produces no results:

MATCH DIFFERENT EDGES () -[E]-> (), () -[E]-> ()

and neither does the following:

MATCH DIFFERENT EDGES () -[E]-> () -[E]-> ()

b) Otherwise, let $BINDINGS$ be $INNER$.

9) A *match* of GPT is a multi-path binding $M = (PB_1, \dots, PB_{NP})$ of NP path bindings in $BINDINGS$, such that all of the following are true:

a) For every j , $1 \text{ (one)} \leq j \leq NP$, and for every <parenthesized path pattern expression> $PPPE$ contained in PP_j , let i be the bracket index of $PPPE$, and let $[_i$ and $]_i$ be the bracket symbols associated with $PPPE$. A *binding* of $PPPE$ is a substring of PB_j that begins with the bracket binding $([_i, [_i)$ and ends with the next bracket binding $(]_i,]_i)$.

NOTE 228 — “Bracket index” is defined in Subclause 22.2, “Machinery for graph pattern matching”.

For every binding *BPPPE* of *PPPE* contained in *PB_j*, all of the following are true:

- i) For every <element variable> *EV* that is exposed as an unconditional singleton by *PPPE*, *EV* is bound to a unique graph element by the element variable bindings contained in *BPPPE*.

NOTE 229 — Anonymous symbols are not <element variable>s; there is no requirement that two anonymous symbols bind to the same graph element.

- ii) If *PPPE* contains a <parenthesized path pattern where clause> *PPPWC*, then *True* is the *VALUE* returned as *V1* when the General Rules of Subclause 22.6, “Application of bindings to evaluate an expression”, with *GPT* as *GRAPH PATTERN*, the <search condition> simply contained in *PPPWC* as *EXPRESSION*, *MACH* as *MACHINERY*, *M* as *MULTI-PATH BINDING*, and a reference to *BPPPE* as *REFERENCE TO LOCAL CONTEXT*.

- b) If *GPT* contains a <graph pattern where clause> *GPWC*, then *True* is the *VALUE* returned as *V2* when the General Rules of Subclause 22.6, “Application of bindings to evaluate an expression”, with *GPT* as *GRAPH PATTERN*, *GPWC* as *EXPRESSION*, *MACH* as *MACHINERY*, *M* as *MULTI-PATH BINDING*, and a reference to *M* as *REFERENCE TO LOCAL CONTEXT*.

- 10) A reduced match *RM* = (*RPB₁*, ..., *RPB_{NP}*) is obtained from a match *M* = (*PB₁*, ..., *PB_{NP}*) as *RM* = *REDUCE*(*M*).

NOTE 230 — Set-theoretic deduplication will occur here. That is, two or more matches can reduce to the same reduced match; this scenario is regarded as contributing only a single reduced match to the result set.

- 11) Let *SRM* be the set of reduced matches.
- 12) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *SRM* as *SET OF REDUCED MATCHES*.

Conformance Rules

- 1) Without Feature G002, “Different-edges match mode”, conforming GQL language shall not contain a <different edges match mode>.
- 2) Without Feature G003, “Explicit REPEATABLE ELEMENTS keyword”, conforming GQL language shall not contain a <match mode> that specifies REPEATABLE ELEMENTS or REPEATABLE ELEMENT BINDINGS.
- 3) Without Feature G004, “Path variables”, conforming GQL language shall not contain a <path pattern> that simply contains a <path variable declaration>.
- 4) Without Feature G005, “Path search prefix in a path pattern”, conforming GQL language shall not contain a <path pattern> that simply contains a <path pattern prefix> that is a <path search prefix>.
- 5) Without Feature G006, “Graph pattern KEEP clause: path mode prefix”, conforming GQL language shall not contain a <keep clause> that simply contains a <path mode prefix>.
- 6) Without Feature G007, “Graph pattern KEEP clause: path search prefix”, conforming GQL language shall not contain a <keep clause> that simply contains a <path search prefix>.

16.5 <insert graph pattern>

Function

Define an <insert graph pattern>.

Format

```
<insert graph pattern> ::=  
  <insert path pattern list>  
  
<insert path pattern list> ::=  
  <insert path pattern> [ { <comma> <insert path pattern> }... ]  
  
<insert path pattern> ::=  
  <insert node pattern> [ { <insert edge pattern> <insert node pattern> }... ]  
  
<insert node pattern> ::=  
  <left paren> [ <insert element pattern filler> ] <right paren>  
  
<insert edge pattern> ::=  
  <insert edge pointing left>  
  | <insert edge pointing right>  
  | <insert edge undirected>  
  
<insert edge pointing left> ::=  
  <left arrow bracket> [ <insert element pattern filler> ] <right bracket minus>  
  
<insert edge pointing right> ::=  
  <minus left bracket> [ <insert element pattern filler> ] <bracket right arrow>  
  
<insert edge undirected> ::=  
  <tilde left bracket> [ <insert element pattern filler> ] <right bracket tilde>  
  
<insert element pattern filler> ::=  
  <element variable declaration> [ <label and property set specification> ]  
  | [ <element variable declaration> ] <label and property set specification>  
  
<label and property set specification> ::=  
  <is or colon> <label set specification> [ <element property specification> ]  
  | [ <is or colon> <label set specification> ] <element property specification>
```

Syntax Rules

- 1) Let *IGP* be the <insert graph pattern>.
- 2) For every <insert element pattern filler> *IEPF* simply contained in *IGP* that does not immediately contain an <element variable declaration>:
 - a) Let *IDN* be the name of a new system-generated regular identifier.
 - b) *IEPF* is effectively replaced by:
IDN IEPF
- 3) The system-generated variable names of *IGP* are the names of all <element variable declaration>s that are system-generated regular identifiers immediately contained in an <insert element pattern filler> simply contained in *IGP*.
- 4) Let *IS* be the <insert statement> that immediately contains *IGP*.

- 5) An <element variable> *EV* contained in an <element variable declaration> *EVD* is said to be declared by *EVD*, and by the <insert node pattern> or <insert edge pattern> *EP* that simply contains *EVD*. The <element variable> specifies the name of an element variable, which is also declared by *EVD* and *EP*.
- 6) For every <value expression> *VE* immediately contained in a <property key value pair> immediately contained in an <element property specification> simply contained in *IGP*:
 - a) The declared type of the incoming working record of *VE* is the declared type of the incoming working record of *IS* amended with the record type of the declared type of the incoming working table of *IS*.
 - b) The declared type of the incoming working table of *VE* is the material unit binding table type.
 - c) The declared type of *VE* shall be a supported property value type.
- 7) An *insert element pattern* is either an <insert node pattern> or an <insert edge pattern> that is simply contained in *IGP*.
- 8) A *bound insert element pattern* is an insert element pattern *IEP* that declares an <element variable> *EV* whose name is either the name of a column *COL* of the declared type of the incoming working table of *IS* or is the name of a field type *FT* of the declared type of the incoming working record of *IS*. The declared type of *IEP* is the column type of *COL* or the value type of *FT*.
- 9) No <property key value pair list> simply contained in a bound insert element pattern shall simply contain two equivalent <property name>s that are each immediately contained in a different <property key value pair>.
- 10) For every <element variable> *EV* simply contained in *IGP*:
 - a) For every bound insert element pattern *EP* that declares *EV*:
 - i) *EP* shall be an <insert node pattern>.
 - ii) The declared type of *EP* shall be a node reference value type.
 - iii) *EP* shall not simply contain a <label and property set specification>.
 - b) The *defining insert element pattern* of *EV* is the first insert element pattern that declares *EV*. Every insert element pattern that declares *EV* and is not the defining insert element pattern of *EV* shall not simply contain a <label and property set specification>.
- 11) If an <insert edge pattern> *EP1* declares an <element variable> *EV1*, then there shall not be an <insert node pattern> or <insert edge pattern> *EP2* that declares an <element variable> *EV2* equivalent to *EV1*.

NOTE 231 — This rule has two consequences.

- If an <insert node pattern> declares an <element variable> *EV*, then there is not an <insert edge pattern> that declares an <element variable> with the same name as *EV*.
- Every <insert edge pattern> is the defining insert element pattern of the element variable it declares.

General Rules

None.

Conformance Rules

- 1) Without Feature GH02, “Undirected edge patterns”, conforming GQL language shall not contain an <insert edge pattern> that is an <insert edge undirected>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

16.6 <path pattern prefix>

Function

Specify a path-finding operation and a path mode.

Format

```
<path pattern prefix> ::=  
    <path mode prefix>  
    | <path search prefix>
```

```
<path mode prefix> ::=  
    <path mode> [ <path or paths> ]
```

```
<path mode> ::=  
    WALK  
    | TRAIL  
    | SIMPLE  
    | ACYCLIC
```

```
<path search prefix> ::=  
    <all path search>  
    | <any path search>  
    | <shortest path search>
```

```
<all path search> ::=  
    ALL [ <path mode> ] [ <path or paths> ]
```

```
<path or paths> ::=  
    PATH | PATHS
```

```
<any path search> ::=  
    ANY [ <number of paths> ] [ <path mode> ] [ <path or paths> ]
```

```
<number of paths> ::=  
    <non-negative integer specification>
```

```
<shortest path search> ::=  
    <all shortest path search>  
    | <any shortest path search>  
    | <counted shortest path search>  
    | <counted shortest group search>
```

```
<all shortest path search> ::=  
    ALL SHORTEST [ <path mode> ] [ <path or paths> ]
```

```
<any shortest path search> ::=  
    ANY SHORTEST [ <path mode> ] [ <path or paths> ]
```

```
<counted shortest path search> ::=  
    SHORTEST <number of paths> [ <path mode> ] [ <path or paths> ]
```

```
<counted shortest group search> ::=  
    SHORTEST [ <number of groups> ] [ <path mode> ] [ <path or paths> ] { GROUP | GROUPS }
```

```
<number of groups> ::=  
    <non-negative integer specification>
```

Syntax Rules

- 1) If a <parenthesized path pattern expression> does not specify a <path mode prefix>, then WALK PATHS is implicit.
- 2) If a <path pattern prefix> *PPP* does not specify <all path search>, then:
 - a) Case:
 - i) If *PPP* does not simply contain a <path mode>, then let *PM* be WALK.
 - ii) Otherwise, let *PM* be the <path mode> simply contained in *PPP*.
 - b) Case:
 - i) If *PPP* does not simply contain a <number of paths> or <number of groups>, then let *N* be an <unsigned integer> whose value is 1 (one).
 - ii) Otherwise, let *N* be the <number of paths> or <number of groups> simply contained in *PPP*. The declared type of *N* shall be exact numeric with scale 0 (zero). If *N* is a <literal>, then the value of *N* shall be positive.
 - c) Case:
 - i) If *PPP* is an <any path search>, then *PPP* is equivalent to:
ANY *N* *PM* PATHS
 - ii) If *PPP* is a <shortest path search>, then
Case:
 - 1) If *PPP* is <all shortest path search>, then *PPP* is equivalent to:
SHORTEST 1 *PM* GROUP
 - 2) If *PPP* is <any shortest path search>, then *PPP* is equivalent to:
SHORTEST 1 *PM* PATH
 - 3) If *PPP* is <counted shortest path search>, then *PPP* is equivalent to:
SHORTEST *N* *PM* PATHS
 - 4) If *PPP* is <counted shortest group search>, then *PPP* is equivalent to:
SHORTEST *N* *PM* GROUPS
- 3) A <path pattern prefix> that specifies a <path mode> other than WALK is *restrictive*. A <parenthesized path pattern expression> that immediately contains a restrictive <path mode prefix> is *restrictive*.
- 4) A <path search prefix> other than <all path search> is *selective*. A <path pattern> that simply contains a selective <path search prefix> is *selective*.
- 5) Let *PPPE* be a selective <path pattern>.
 - a) An element variable exposed by *PPPE* is an *interior variable* of *PPPE*.
 - b) A node variable *LVV* is the *left boundary variable* of *PPPE* if all of the following conditions are true:
 - i) *PPPE* exposes *LVV* as an unconditional singleton variable.

- ii) *LVP* is declared in the first explicit or implicit <node pattern> *LVP* contained in *PPPE*.
 - iii) *LVP* is not contained in a <path pattern union> or <path multiset alternation> that is contained in *PPPE*.
 - c) A node variable *RVV* is the *right boundary variable* of *PPPE* if all of the following conditions are true:
 - i) *PPPE* exposes *RVV* as an unconditional singleton variable.
 - ii) *RVV* is declared in the last explicit or implicit <node pattern> *RVP* contained in *PPPE*.
 - iii) *RVP* is not contained in a <path pattern union> or <path multiset alternation> that is contained in *PPPE*.
 - d) An element variable that is exposed by *PPPE* that is neither a left boundary variable of *PPPE* nor a right boundary variable of *PPPE* is a *strict interior variable* of *PPPE*.
- 6) An element variable that is not declared in a selective <path pattern> is an *exterior variable*.
- 7) A strict interior variable of one selective <path pattern> shall not be equivalent to an exterior variable, or to an interior variable of another selective <path pattern>.
- NOTE 232 — This does not prohibit implicit joins of boundary variables of selective <path pattern>s with exterior variables or boundary variables of other selective <path pattern>s.
- 8) A selective <path pattern> *SPP* shall not contain a reference to a graph pattern variable that is not declared by *SPP*.
- NOTE 233 — This rule, and the prohibition of implicit joins to exterior variables and interior variables of other selective <path pattern>s, insure that each selective <path pattern> can be evaluated in isolation from any other <path pattern>.

General Rules

None.

NOTE 234 — Restrictive <path mode>s are enforced as part of the check for consistent path bindings in the generation of the set of local matches in Subclause 16.7, "<path pattern expression>". Selective <path pattern>s are evaluated by Subclause 22.4, "Evaluation of a selective <path pattern>".

Conformance Rules

- 1) Without Feature G010, "Explicit WALK keyword", conforming GQL language shall not contain a <path mode> that specifies WALK.
- 2) Without Feature G011, "Advanced path modes: TRAIL", conforming GQL language shall not contain a <path mode> that specifies TRAIL.
- 3) Without Feature G012, "Advanced path modes: SIMPLE", conforming GQL language shall not contain a <path mode> that specifies SIMPLE.
- 4) Without Feature G013, "Advanced path modes: ACYCLIC", conforming GQL language shall not contain a <path mode> that specifies ACYCLIC.
- 5) Without Feature G014, "Explicit PATH/PATHS keywords", conforming GQL language shall not contain a <path or paths>.
- 6) Without Feature G015, "All path search: explicit ALL keyword", conforming GQL language shall not contain an <all path search>.

ISO/IEC 39075:2024(en)
16.6 <path pattern prefix>

- 7) Without Feature G016, “Any path search”, conforming GQL language shall not contain an <any path search>.
- 8) Without Feature G017, “All shortest path search”, conforming GQL language shall not contain an <all shortest path search>.
- 9) Without Feature G018, “Any shortest path search”, conforming GQL language shall not contain an <any shortest path search>.
- 10) Without Feature G019, “Counted shortest path search”, conforming GQL language shall not contain a <counted shortest path search>.
- 11) Without Feature G020, “Counted shortest group search”, conforming GQL language shall not contain a <counted shortest group search>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

16.7 <path pattern expression>

Function

Specify a pattern to match a single path in a property graph.

Format

```
<path pattern expression> ::=
    <path term>
  | <path multiset alternation>
  | <path pattern union>

<path multiset alternation> ::=
    <path term> <multiset alternation operator> <path term>
    [ { <multiset alternation operator> <path term> }... ]

<path pattern union> ::=
    <path term> <vertical bar> <path term> [ { <vertical bar> <path term> }... ]

<path term> ::=
    <path factor>
  | <path concatenation>

<path concatenation> ::=
    <path term> <path factor>

<path factor> ::=
    <path primary>
  | <quantified path primary>
  | <questioned path primary>

<quantified path primary> ::=
    <path primary> <graph pattern quantifier>

<questioned path primary> ::=
    <path primary> <question mark>

NOTE 235 — Unlike most regular expression languages, <question mark> is not equivalent to the quantifier
{0,1}: the quantifier {0,1} exposes variables as group, whereas <question mark> does not change the singleton
variables that it exposes to group. However, <question mark> does expose any singleton variables as conditional
singletons.

<path primary> ::=
    <element pattern>
  | <parenthesized path pattern expression>
  | <simplified path pattern expression>

<element pattern> ::=
    <node pattern>
  | <edge pattern>

<node pattern> ::=
    <left paren> <element pattern filler> <right paren>

<element pattern filler> ::=
    [ <element variable declaration> ]
    [ <is label expression> ]
    [ <element pattern predicate> ]

<element variable declaration> ::=
```

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

```
[ TEMP ] <element variable>

<is label expression> ::=
  <is or colon> <label expression>

<is or colon> ::=
  IS
  | <colon>

<element pattern predicate> ::=
  <element pattern where clause>
  | <element property specification>

<element pattern where clause> ::=
  WHERE <search condition>

<element property specification> ::=
  <left brace> <property key value pair list> <right brace>

<property key value pair list> ::=
  <property key value pair> [ { <comma> <property key value pair> }... ]

<property key value pair> ::=
  <property name> <colon> <value expression>

<edge pattern> ::=
  <full edge pattern>
  | <abbreviated edge pattern>

<full edge pattern> ::=
  <full edge pointing left>
  | <full edge undirected>
  | <full edge pointing right>
  | <full edge left or undirected>
  | <full edge undirected or right>
  | <full edge left or right>
  | <full edge any direction>

<full edge pointing left> ::=
  <left arrow bracket> <element pattern filler> <right bracket minus>

<full edge undirected> ::=
  <tilde left bracket> <element pattern filler> <right bracket tilde>

<full edge pointing right> ::=
  <minus left bracket> <element pattern filler> <bracket right arrow>

<full edge left or undirected> ::=
  <left arrow tilde bracket> <element pattern filler> <right bracket tilde>

<full edge undirected or right> ::=
  <tilde left bracket> <element pattern filler> <bracket tilde right arrow>

<full edge left or right> ::=
  <left arrow bracket> <element pattern filler> <bracket right arrow>

<full edge any direction> ::=
  <minus left bracket> <element pattern filler> <right bracket minus>

<abbreviated edge pattern> ::=
  <left arrow>
  | <tilde>
  | <right arrow>
  | <left arrow tilde>
  | <tilde right arrow>
```

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

| <left minus right>
| <minus sign>

<parenthesized path pattern expression> ::=
 <left paren>
 [<subpath variable declaration>]
 [<path mode prefix>]
 <path pattern expression>
 [<parenthesized path pattern where clause>]
 <right paren>

<subpath variable declaration> ::=
 <subpath variable> <equals operator>

<parenthesized path pattern where clause> ::=
 WHERE <search condition>

Syntax Rules

- 1) Let *GP* be the outermost <graph pattern> that contains the <path pattern expression>.
- 2) Let *RIGHTMINUS* be the following collection of <token>s: <right bracket minus>, <left arrow>, <slash minus>, and <minus sign>.
NOTE 236 — These are the tokens ("]-", "<-", "/-", and "-") that expose a minus sign on the right.
- 3) Let *LEFTMINUS* be the following collection of <token>s: <minus left bracket>, <right arrow>, <minus slash>, and <minus sign>.
NOTE 237 — These are the tokens ("[-", "<->", "-/", and "-") that expose a minus sign on the left. <minus sign> itself is in both *RIGHTMINUS* and *LEFTMINUS*.
- 4) A <path pattern expression> shall not juxtapose a <token> from *RIGHTMINUS* followed by a <token> from *LEFTMINUS* without a <separator> between them.
NOTE 238 — Otherwise, the concatenation of the two tokens would include the sequence of two <minus sign>s, which is a <simple comment introducer>.
- 5) A <path pattern expression> that contains at the same depth of graph pattern matching a variable quantifier, a <questioned path primary>, a <path multiset alternation>, or a <path pattern union> is a *possibly variable length path pattern*.
- 6) A <path pattern expression> that is not a possibly variable length path pattern is a *fixed length path pattern*.
- 7) The *minimum path length* of certain BNF non-terminal instances defined in this Subclause is defined recursively as follows:
 - a) The minimum path length of a <node pattern> is 0 (zero).
 - b) The minimum path length of an <edge pattern> is 1 (one).
 - c) The minimum path length of a <path concatenation> is the sum of the minimum path lengths of its operands.
 - d) The minimum path length of a <path pattern union> or <path multiset alternation> is the minimum of the minimum path length of its operands.
 - e) The minimum path length of a <quantified path primary> is the product of the minimum path length of the simply contained <path primary> and the value of the <lower bound>.
 - f) The minimum path length of a <questioned path primary> is 0 (zero).

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

- g) The minimum path length of a <parenthesized path pattern expression> is the minimum path length of the simply contained <path pattern expression>.
- h) If $BNT1$ and $BNT2$ are two BNF non-terminal instances such that $BNT1 ::= BNT2$ and the minimum path length of $BNT2$ is defined, then the minimum path length of $BNT1$ is also defined and is the same as the minimum path length of $BNT2$.
- 8) The <path primary> immediately contained in a <quantified path primary> or <questioned path primary> shall have minimum path length that is greater than 0 (zero).
- 9) The <path primary> simply contained in a <quantified path primary> shall not contain a <quantified path primary> at the same depth of graph pattern matching.
- 10) Let PMA be a <path multiset alternation>.
 - a) A <path term> simply contained in PMA is a *multiset alternation operand* of PMA .
 - b) Let $NOPMA$ be the number of multiset alternation operands of PMA . Let $OPMA_1, \dots, OPMA_{NOPMA}$ be an enumeration of the operands of PMA .
 - c) Any <subpath variable>s declared by <subpath variable declaration>s simply contained in the multiset alternation operands of PMA shall be mutually distinct.
 - d) Let $SOPMA_1, \dots, SOPMA_{NOPMA}$ be new system-generated regular identifiers.
 - e) For i , $1 \text{ (one)} \leq i \leq NOPMA$,
 Case:
 - i) If $OPMA_i$ is a <parenthesized path pattern expression> that simply contains a <subpath variable declaration>, then let $OPMAX_i$ be $OPMA_i$.
 - ii) Otherwise, let $OPMAX_i$ be the <parenthesized path pattern expression>
 $(SOPMA_i = OPMA_i)$
 - f) PMA is equivalent to:
 $OPMAX_1 \mid \dots \mid OPMAX_{NOPMA}$
- 11) A <path term> $PPUOP$ simply contained in a <path pattern union> PSD is a *path pattern union operand* of PSD .
 $PPUOP$ shall not contain a reference to an element variable that is not declared in $PPUOP$ or outside of PSD .
- 12) An <element pattern> EP that contains an <element pattern where clause> $EPWC$ is transformed as follows:
 - a) Let EPF be the <element pattern filler> simply contained in EP .
 - b) Let $PREFIX$ be the <delimiter token> contained in EP before EPF and let $SUFFIX$ be the <delimiter token> contained in EP after EPF .
 - c) Let EV be the <element variable> simply contained in EPF .
 - d) Let ILE be defined as follows. If EPF simply contains the <is label expression> ILE_CAND , then ILE is ILE_CAND ; otherwise, ILE is the zero-length character string.
 - e) EP is effectively replaced by
 $(PREFIX EV ILE SUFFIX EPWC)$

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

- 13) An <element pattern> that does not contain an <element variable declaration>, an <is label expression>, or an <element pattern predicate> is said to be *empty*.
- 14) Each <path pattern expression> is transformed in the following steps:
- a) If the <path primary> immediately contained in a <quantified path primary> or <questioned path primary> is an <edge pattern> *EP*, then *EP* is effectively replaced by
 (*EP*)
 NOTE 239 — For example,
 ->*
 becomes:
 (->) {0,}
 which in later transformations becomes:
 (() -> ()) {0,}
 - b) If two successive <element pattern>s contained in a <path concatenation> at the same depth of graph pattern matching are <edge pattern>s, then an implicit empty <node pattern> is inserted between them.
 - c) If an edge pattern *EP* contained in a <path term> *PST* at the same depth of graph pattern matching is not preceded by a <node pattern> contained in *PST* at the same depth of graph pattern matching, then an implicit empty <node pattern> is inserted in *PST* immediately prior to *EP*.
 - d) If an edge pattern *EP* contained in a <path term> *PST* at the same depth of graph pattern matching is not followed by a <node pattern> contained in *PST* at the same depth of graph pattern matching, then an implicit empty <node pattern> is inserted in *PST* immediately after *EP*.
 NOTE 240 — As a result of the preceding transformations, a fixed length path pattern has an odd number of <element pattern>s, beginning and ending with <node pattern>s, and alternating between <node pattern>s and <edge pattern>s.
 - e) Every <abbreviated edge pattern> *AEP* is effectively replaced with an empty <full edge pattern> as follows.
 Case:
 - i) If *AEP* is <left arrow>, then *AEP* is effectively replaced by the <full edge pointing left>:
 <- []-
 - ii) If *AEP* is <tilde>, then *AEP* is effectively replaced by the <full edge undirected>:
 ~[]~
 - iii) If *AEP* is <right arrow>, then *AEP* is effectively replaced by the <full edge pointing right>:
 -[]->
 - iv) If *AEP* is <left arrow tilde>, then *AEP* is effectively replaced by the <full edge left or undirected>:
 <~[]~

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

- v) If *AEP* is <tilde right arrow>, then *AEP* is effectively replaced by the <full edge undirected or right>:
~[]~>
- vi) If *AEP* is <left minus right>, then *AEP* is effectively replaced by the <full edge left or right>:
<- [] ->
- vii) If *AEP* is <minus sign>, then *AEP* is effectively replaced by the <full edge any direction>:
-[]-

15) The *minimum node count* of certain BNF non-terminal instances defined in this Subclause is defined recursively as follows:

- a) The minimum node count of a <node pattern> is 1 (one).
- b) The minimum node count of an <edge pattern> is 0 (zero).
- c) The minimum node count of a <path concatenation> *PC* is:
Case:
 - i) If two successive <element pattern>s contained in *PC* at the same depth of graph pattern matching are <node pattern>s, then 1 (one) less than the sum of the minimum node counts of its operands.
 - ii) Otherwise, the sum of the minimum node counts of its operands.
- d) The minimum node count of a <path pattern union> or <path multiset alternation> is the minimum of the minimum node count of its operands.
- e) The minimum node count of a <quantified path primary> is the product of the minimum node count of the simply contained <path primary> and the value of the <lower bound> of the simply contained <graph pattern quantifier>.
- f) The minimum node count of a <questioned path primary> is 0 (zero).
- g) The minimum node count of a <parenthesized path pattern expression> is the minimum node count of the simply contained <path pattern expression>.
- h) If *BNF1* and *BNF2* are two BNF non-terminal instances such that *BNF1* ::= *BNF2* and the minimum node count of *BNF2* is defined, then the minimum node count of *BNF1* is also defined and is the same as the minimum node count of *BNF2*.

16) The <path pattern expression> simply contained in a <path pattern> shall have a minimum node count that is greater than 0 (zero).

NOTE 241 — The minimum node count is computed after the syntactic transform that adds implicit node patterns. Thus a single <edge pattern> is a permitted <path pattern> because it implies two <node pattern>s.

NOTE 242 — A later Syntax Rule makes the same requirement of a <parenthesized path pattern expression> that simply contains a <subpath variable declaration>.

17) An <element variable> *EV* contained in an <element variable declaration> *GPVD* is said to be *declared* by *GPVD*, and by the <element pattern> *EP* that simply contains *GPVD*. The <element variable> is the name of an element variable, which is also declared by *GPVD* and *EP*. If *GPVD* simply contains TEMP, then *EV* is a temporary element variable.

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

NOTE 243 — Element bindings to temporary element variables are removed prior to set-theoretic deduplication of matches. See [General Rule 10](#)) of Subclause 16.4, “<graph pattern>” and [General Rule 14](#)) of Subclause 22.2, “Machinery for graph pattern matching”.

- 18) An element variable that is declared by a <node pattern> is a node variable. An element variable that is declared by an <edge pattern> is an edge variable.
- 19) The scope of an <element variable> *EV* that is declared by an <element pattern> *EP* is defined as follows. If *EV* is a temporary element variable, then the scope of *EV* is the innermost <path term> containing *EP*; otherwise, the scope of *EV* is the innermost <graph pattern binding table> containing *EP*.
- 20) If a <parenthesized path pattern expression> *PPPE* simply contains a <subpath variable declaration>, then the minimum node count of *PPPE* shall be greater than 0 (zero).
- 21) If an <element pattern> *EP* that contains an <element pattern where clause> *EPWC*, then *EP* shall simply contain an <element variable declaration> *GPVD*.
- 22) If *EV* is an element variable or subpath variable, and *BNT* is an instance of a BNF non-terminal, then the terminology “*BNT* exposes *EV*” is defined as follows. The full terminology is one of the following: “*BNT* exposes *EV* as an unconditional singleton variable”, “*BNT* exposes *EV* as a conditional singleton variable”, “*BNT* exposes *EV* as an effectively bounded group variable” or “*BNT* exposes *EV* as an effectively unbounded group variable”. The terms “unconditional singleton variable”, “conditional singleton variable”, “effectively bounded group variable”, and “effectively unbounded group variable” are called the *degree of exposure*.
 - a) An <element pattern> *EP* that declares an element variable *EV* exposes *EV* as an unconditional singleton.
 - b) A <parenthesized path pattern expression> *PPPE* that simply contains a <subpath variable declaration> that declares *EV* exposes *EV* as an unconditional singleton variable. *PPPE* shall not contain another <parenthesized path pattern expression> that declares *EV*.
 - c) If a <path concatenation> *PPC* declares *EV* then let *PT* be the <path term> and let *PF* be the <path factor> simply contained in *PPC*.

Case:

 - i) If *EV* is exposed as an unconditional singleton by both *PT* and *PF*, then *EV* is exposed as an unconditional singleton by *PPC*. *EV* shall not be a subpath variable.

NOTE 244 — This case expresses an implicit join on *EV* within *PPC*. Implicit joins between conditional singleton variables, group variables, or subpath variables are forbidden.
 - ii) Otherwise, *EV* shall only be exposed by one of *PT* or *PF*. In this case *EV* is exposed by *PPC* in the same degree that it is exposed by *PT* or *PF*.
 - d) If a <path pattern union> or <path multiset alternation> *PA* declares *EV*, then

Case:

 - i) If every operand of *PA* exposes *EV* as an unconditional singleton variable, then *PA* exposes *EV* as an unconditional singleton variable.
 - ii) If at least one operand of *PA* exposes *EV* as an effectively unbounded group variable, then *PA* exposes *EV* as an effectively unbounded group variable.
 - iii) If at least one operand of *PA* exposes *EV* as an effectively bounded group variable, then *PA* exposes *EV* as an effectively bounded group variable.
 - iv) Otherwise, *PA* exposes *EV* as a conditional singleton variable.

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

- e) If a <quantified path primary> *QPP* declares *EV*, then let *PP* be the <path primary> simply contained in *QPP*.

Case:

- i) If *QPP* contains a <graph pattern quantifier> that is a <fixed quantifier> or a <general quantifier> that contains an <upper bound> and *PP* does not expose *EV* as an effectively unbounded group variable, then *QPP* exposes *EV* as an effectively bounded group variable.
- ii) If *QPP* is contained at the same depth of graph pattern matching in a restrictive <parenthesized path pattern expression>, then *QPP* exposes *EV* as an effectively bounded group variable.

NOTE 245 — The preceding definition is applied after the syntactic transformation to insure that every <path mode prefix> is at the head of a <parenthesized path pattern expression>.

- iii) Otherwise, *QPP* exposes *EV* as an effectively unbounded group variable.

- f) If a <questioned path primary> *QUPP* declares *EV*, then let *PP* be the <path primary> simply contained in *QUPP*.

Case:

- i) If *PP* exposes *EV* as a group variable, then *QUPP* exposes *EV* as a group variable with the same degree of exposure.
- ii) Otherwise, *QUPP* exposes *EV* as a conditional singleton variable.

- g) A <parenthesized path pattern expression> exposes the same variables as the simply contained <path pattern expression>, in the same degree of exposure.

NOTE 246 — A restrictive <path mode> declared by a <parenthesized path pattern expression> makes variables effectively bounded, but it does so even for proper subexpressions within the scope of the <path mode> and has already been handled by the rules for <quantified path primary>.

- h) If a <path pattern> *PP* declares *EV*, then let *PPE* be the simply contained <path pattern expression>.

Case:

- i) If *PPE* exposes *EV* as an unconditional singleton, a conditional singleton, or an effectively bounded group variable, then *PP* exposes *EV* with the same degree of exposure.
- ii) Otherwise, *PP* exposes *EV* as an effectively bounded group variable.

NOTE 247 — That is, even if *PPE* exposes *EV* as an effectively unbounded group variable, *PP* still exposes *EV* as effectively bounded, because in this case *PP* is either required to be a selective <path pattern> or to be in the scope of the <different edges match mode>.

- i) If *BNT1* and *BNT2* are two BNF non-terminal instances such that *BNT1* ::= *BNT2* and *BNT2* exposes *EV*, then *BNT1* exposes *EV* to the same degree of exposure as *BNT2*.

- 23) If *BNT* is a BNF non-terminal instance that exposes a graph pattern variable *GPV* with a degree of exposure *DEGREE*, then *BNT* is also said to expose the name of *GPV* with degree of exposure *DEGREE*.
- 24) A <parenthesized path pattern where clause> simply contained in a <parenthesized path pattern expression> *PPPE* shall not reference a path variable.

General Rules

None.

NOTE 248 — The evaluation of a <path pattern expression> is performed by the General Rules of [Subclause 22.3](#), “Evaluation of a <path pattern expression>”.

Conformance Rules

- 1) Conforming GQL language shall not contain an <element variable declaration> that immediately contains TEMP.
NOTE 249 — An <element variable declaration> containing TEMP is a specification device and is not syntax available to the user.
- 2) Without Feature G030, “Path multiset alternation”, conforming GQL language shall not contain a <path multiset alternation>.
- 3) Without Feature G031, “Path multiset alternation: variable length path operands”, in conforming GQL language, an operand of a <path multiset alternation> shall be a fixed length path pattern.
- 4) Without Feature G032, “Path pattern union”, conforming GQL language shall not contain a <path pattern union>.
- 5) Without Feature G033, “Path pattern union: variable length path operands”, in conforming GQL language, an operand of a <path pattern union> shall be a fixed length path pattern.
- 6) Without Feature G035, “Quantified paths”, conforming GQL language shall not contain a <quantified path primary> that does not immediately contain a <path primary> that is an <edge pattern>.
- 7) Without Feature G036, “Quantified edges”, conforming GQL language shall not contain a <quantified path primary> that immediately contains a <path primary> that is an <edge pattern>.
- 8) Without Feature G037, “Questioned paths”, conforming GQL language shall not contain a <questioned path primary>.
- 9) Without Feature G038, “Parenthesized path pattern expression”, conforming GQL language shall not contain a <parenthesized path pattern expression>.
- 10) Without Feature G041, “Non-local element pattern predicates”, in conforming GQL language, the <element pattern where clause> of an <element pattern> *EP* shall only reference the <element variable> declared in *EP*.
- 11) Without Feature G043, “Complete full edge patterns”, conforming GQL language shall not contain a <full edge pattern> that is not a <full edge any direction>, a <full edge pointing left>, or a <full edge pointing right>.
- 12) Without Feature G044, “Basic abbreviated edge patterns”, conforming GQL language shall not contain an <abbreviated edge pattern> that is a <minus sign>, a <left arrow>, or a <right arrow>.
- 13) Without Feature G045, “Complete abbreviated edge patterns”, conforming GQL language shall not contain an <abbreviated edge pattern> that is not a <minus sign>, a <left arrow>, or a <right arrow>.
- 14) Without Feature G046, “Relaxed topological consistency: adjacent vertex patterns”, in conforming GQL language, between any two <node pattern>s contained in a <path pattern expression> there shall be at least one <edge pattern>, <left paren>, or <right paren>.
- 15) Without Feature G047, “Relaxed topological consistency: concise edge patterns”, in conforming GQL language, an <edge pattern> shall be immediately preceded and followed by a <node pattern>.
- 16) Without Feature G048, “Parenthesized path pattern: subpath variable declaration”, conforming GQL language shall not contain a <parenthesized path pattern expression> that simply contains a <subpath variable declaration>.

ISO/IEC 39075:2024(en)
16.7 <path pattern expression>

- 17) Without Feature G049, “Parenthesized path pattern: path mode prefix”, conforming GQL language shall not contain a <parenthesized path pattern expression> that immediately contains a <path mode prefix>.
- 18) Without Feature G050, “Parenthesized path pattern: WHERE clause”, conforming GQL language shall not contain a <parenthesized path pattern where clause>.
- 19) Without Feature G051, “Parenthesized path pattern: non-local predicates”, in conforming GQL language, a <parenthesized path pattern where clause> simply contained in a <parenthesized path pattern expression> *PPPE* shall not reference an <element variable> that is not declared in *PPPE*.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

16.8 <label expression>

Function

Specify an expression that matches one or more labels of a graph.

Format

```
<label expression> ::=  
    <label term>  
    | <label disjunction>  
  
<label disjunction> ::=  
    <label expression> <vertical bar> <label term>  
  
<label term> ::=  
    <label factor>  
    | <label conjunction>  
  
<label conjunction> ::=  
    <label term> <ampersand> <label factor>  
  
<label factor> ::=  
    <label primary>  
    | <label negation>  
  
<label negation> ::=  
    <exclamation mark> <label primary>  
  
<label primary> ::=  
    <label name>  
    | <wildcard label>  
    | <parenthesized label expression>  
  
<wildcard label> ::=  
    <percent>  
  
<parenthesized label expression> ::=  
    <left paren> <label expression> <right paren>
```

Syntax Rules

- 1) Let *LE* be the <label expression>.
- 2) The current working graph site of *LE* shall not be “omitted”.
- 3) Let *CWGS* be the current working graph site of *LE*.
- 4) Case:
 - a) If *LE* is simply contained in an <is label expression>, then:
 - i) Let *GRVT* be the graph reference value type that is the declared type of *CWGS*.
 - ii) If *GRVT* is a closed graph reference value type with constraining GQL-object type *COT* then:
 - 1) If *LE* is simply contained in a <node pattern>, then *LE* is a *node* <label expression>. Every <label name> contained in *LE* shall identify a node label of *COT*.

- 2) Otherwise, *LE* is an *edge* <label expression>. Every <label name> contained in *LE* shall identify an edge label of *COT*.
- b) Otherwise, *LE* is simply contained in a <labeled predicate> *LP* and then:
 - i) Let *EVR* be the <element variable reference> in *LP*.
 - ii) If the declared type of *EVR* is a closed graph reference value type with constraining GQL-object type *COT*, then every <label name> contained in *LE* shall identify a label in the label set of *COT*.

NOTE 250 — *COT* is either a node type or an edge type. Subclause 19.9, “<labeled predicate>”, Syntax Rule 2), requires that *EVR* has singleton degree of reference and in this case, Subclause 16.10, “<element variable reference>”, Syntax Rule 3)a) requires that *EVR* is a node reference value or an edge reference value type, whose constraining GQL-object type is a node type or an edge type, respectively.

General Rules

None.

Conformance Rules

- 1) Without Feature G074, “Label expression: wildcard label”, conforming GQL language shall not contain a <wildcard label>.

16.9 <path variable reference>

Function

Specify path variable references.

Format

<path variable reference> ::=
 <binding variable reference>

Syntax Rules

- 1) Let *PVR* be the <path variable reference>.
- 2) The degree of reference of *PVR* shall be unconditional singleton.
- 3) The declared type of *PVR* shall be a path value type.

General Rules

None.

NOTE 251 — Every <path variable reference> is evaluated in the General Rules of Subclause 20.12, “<binding variable reference>” by looking it up in the current working table. However the actual path value is originally constructed and bound in the General Rules of Subclause 22.6, “Application of bindings to evaluate an expression”.

Conformance Rules

None.

16.10 <element variable reference>

Function

Specify element variable references.

Format

```
<element variable reference> ::=  
  <binding variable reference>
```

Syntax Rules

- 1) Let *EVR* be the <element variable reference>.
- 2) Let *EVRDT* be the declared type of *EVR*.
- 3) Case:
 - a) If the degree of reference of *EVR* is singleton, then *EVRDT* shall be a node reference value type or an edge reference value type.
 - b) Otherwise, the degree of reference of *EVR* is group and *EVRDT* shall be a group list value type whose list element type only includes node reference values, edge reference values, or the null value.

General Rules

None.

NOTE 252 — Every <element variable reference> is evaluated in the General Rules of Subclause 20.12, “<binding variable reference>” by looking it up in the current working record. However the actual element variable reference value is originally constructed and bound in the General Rules of Subclause 22.6, “Application of bindings to evaluate an expression”.

Conformance Rules

None.

16.11 <graph pattern quantifier>

Function

Specify a graph pattern quantifier.

Format

```
<graph pattern quantifier> ::=  
    <asterisk>  
    | <plus sign>  
    | <fixed quantifier>  
    | <general quantifier>  
  
<fixed quantifier> ::=  
    <left brace> <unsigned integer> <right brace>  
  
<general quantifier> ::=  
    <left brace> [ <lower bound> ] <comma> [ <upper bound> ] <right brace>  
  
<lower bound> ::=  
    <unsigned integer>  
  
<upper bound> ::=  
    <unsigned integer>
```

Syntax Rules

- 1) The maximum value of <upper bound> is implementation-defined (IL018). <upper bound>, if specified, shall not be greater than this value.
- 2) Every <graph pattern quantifier> is normalized, as follows:
 - a) <asterisk> is equivalent to:
 $\{0, \}$
 - b) <plus sign> is equivalent to:
 $\{1, \}$
 - c) If <fixed quantifier> FQ is specified, then let UI be the <unsigned integer> contained in FQ . FQ is equivalent to:
 $\{UI, UI\}$
 - d) If <general quantifier> GQ is specified, and if <lower bound> is not specified, then the <unsigned integer> 0 (zero) is supplied as the <lower bound>.
- 3) If <general quantifier> GQ is specified or implied by the preceding normalizations, then
Case:
 - a) If <upper bound> is specified, then:
 - i) The value of <upper bound> VUP shall be greater than 0 (zero).
 - ii) The value of <lower bound> LUP shall be less than or equal to VUP .

- iii) If *LUP* equals *VUP*, then *GQ* is a *fixed quantifier*.
 - iv) *GQ* is a *bounded quantifier*.
 - b) Otherwise, *GQ* is an *unbounded quantifier*.
- 4) A <graph pattern quantifier> that is not a fixed quantifier is a *variable quantifier*.

General Rules

None.

Conformance Rules

- 1) Without Feature G060, “Bounded graph pattern quantifiers”, conforming GQL language shall not contain a <fixed quantifier> or a <general quantifier> that immediately contains an <upper bound>.
- 2) Without Feature G061, “Unbounded graph pattern quantifiers”, conforming GQL language shall not contain a <graph pattern quantifier> that immediately contains an <asterisk>, a <plus sign>, or a <general quantifier> that does not immediately contain an <upper bound>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

16.12 <simplified path pattern expression>

Function

Express a path pattern as a regular expression of edge labels.

Format

```
<simplified path pattern expression> ::=
  <simplified defaulting left>
  | <simplified defaulting undirected>
  | <simplified defaulting right>
  | <simplified defaulting left or undirected>
  | <simplified defaulting undirected or right>
  | <simplified defaulting left or right>
  | <simplified defaulting any direction>

<simplified defaulting left> ::=
  <left minus slash> <simplified contents> <slash minus>

<simplified defaulting undirected> ::=
  <tilde slash> <simplified contents> <slash tilde>

<simplified defaulting right> ::=
  <minus slash> <simplified contents> <slash minus right>

<simplified defaulting left or undirected> ::=
  <left tilde slash> <simplified contents> <slash tilde>

<simplified defaulting undirected or right> ::=
  <tilde slash> <simplified contents> <slash tilde right>

<simplified defaulting left or right> ::=
  <left minus slash> <simplified contents> <slash minus right>

<simplified defaulting any direction> ::=
  <minus slash> <simplified contents> <slash minus>

<simplified contents> ::=
  <simplified term>
  | <simplified path union>
  | <simplified multiset alternation>

<simplified path union> ::=
  <simplified term> <vertical bar> <simplified term>
  [ { <vertical bar> <simplified term> }... ]

<simplified multiset alternation> ::=
  <simplified term> <multiset alternation operator> <simplified term>
  [ { <multiset alternation operator> <simplified term> }... ]

<simplified term> ::=
  <simplified factor low>
  | <simplified concatenation>

<simplified concatenation> ::=
  <simplified term> <simplified factor low>

<simplified factor low> ::=
  <simplified factor high>
  | <simplified conjunction>
```

ISO/IEC 39075:2024(en)
16.12 <simplified path pattern expression>

```
<simplified conjunction> ::=
  <simplified factor low> <ampersand> <simplified factor high>

<simplified factor high> ::=
  <simplified tertiary>
  | <simplified quantified>
  | <simplified questioned>

<simplified quantified> ::=
  <simplified tertiary> <graph pattern quantifier>

<simplified questioned> ::=
  <simplified tertiary> <question mark>

<simplified tertiary> ::=
  <simplified direction override>
  | <simplified secondary>

<simplified direction override> ::=
  <simplified override left>
  | <simplified override undirected>
  | <simplified override right>
  | <simplified override left or undirected>
  | <simplified override undirected or right>
  | <simplified override left or right>
  | <simplified override any direction>

<simplified override left> ::=
  <left angle bracket> <simplified secondary>

<simplified override undirected> ::=
  <tilde> <simplified secondary>

<simplified override right> ::=
  <simplified secondary> <right angle bracket>

<simplified override left or undirected> ::=
  <left arrow tilde> <simplified secondary>

<simplified override undirected or right> ::=
  <tilde> <simplified secondary> <right angle bracket>

<simplified override left or right> ::=
  <left angle bracket> <simplified secondary> <right angle bracket>

<simplified override any direction> ::=
  <minus sign> <simplified secondary>

<simplified secondary> ::=
  <simplified primary>
  | <simplified negation>

<simplified negation> ::=
  <exclamation mark> <simplified primary>

<simplified primary> ::=
  <label name>
  | <left paren> <simplified contents> <right paren>
```

Syntax Rules

- 1) A <simplified negation> shall not contain a <simplified concatenation>, <simplified quantified>, <simplified questioned>, or <simplified multiset alternation>.

ISO/IEC 39075:2024(en)
16.12 <simplified path pattern expression>

- 2) A <simplified direction override> shall not contain another <simplified direction override>.
- 3) A <simplified direction override> shall not contain <simplified concatenation>, <simplified quantified>, <simplified questioned>, or <simplified multiset alternation>.
- 4) A <simplified conjunction> shall not contain a <simplified concatenation>, <simplified quantified>, <simplified questioned>, or <simplified multiset alternation>.
- 5) A <simplified path pattern expression> *SPPE* is effectively replaced by:

(*SPPE*)

NOTE 253 — This is done once for each <simplified path pattern expression> prior to the following recursive transformation and not with each iteration of the transformation.

- 6) The following rules are recursively applied until no <simplified path pattern expression>s remain.

NOTE 254 — The rules work from the root of the parse tree of a <simplified path pattern expression>. At each step, the coarsest analysis of a <simplified path pattern expression> is replaced, eliminating at least one level of the parse tree, measured from the root. Note that each replacement can create more <simplified path pattern expression>s than before, but these replacements have less depth. Eventually the recursion replaces <simplified path pattern expression> with <edge pattern>.

- a) Let *SPPE* be a <simplified path pattern expression>.
 - i) Let *SC* be the <simplified contents> contained in *SPPE*.
 - ii) Let *PREFIX* be the <minus slash>, <left minus slash>, <tilde slash>, <left tilde slash>, or <left minus slash> contained in *SPPE*.
 - iii) Let *SUFFIX* be the <slash minus right>, <slash minus>, <slash tilde>, or <slash tilde right> contained in *SPPE*.
 - iv) Let *EDGEPRE* and *EDGESUF* be determined by Table 3, “Conversion of simplified syntax delimiters to default edge delimiters”, from the row containing the values of *PREFIX* and *SUFFIX*.

Table 3 — Conversion of simplified syntax delimiters to default edge delimiters

<i>PREFIX</i>	<i>SUFFIX</i>	<i>EDGEPRE</i>	<i>EDGESUF</i>
- /	/ ->	- [	] ->
< - /	/ -	< - [	] -
~ /	/ ~	~ [	] ~
~ /	/ ~>	~ [	] ~>
< ~ /	/ ~	< ~ [	] ~
< - /	/ ->	< - [	] ->
- /	/ -	- [	] -

- b) Case:

ISO/IEC 39075:2024(en)
16.12 <simplified path pattern expression>

- i) If *SC* is a <simplified path union> *SPU*, then let *N* be the number of <simplified term>s simply contained in *SPU*, and let *ST*₁, ..., *ST*_{*N*} be those <simplified term>s; *SPPE* is effectively replaced by:

*PREFIX ST*₁ *SUFFIX* | *PREFIX ST*₂ *SUFFIX* | ... | *PREFIX ST*_{*N*} *SUFFIX*

- ii) If *SC* is a <simplified multiset alternation> *SMA*, then let *N* be the number of <simplified term>s simply contained in *SMA*, and let *ST*₁, ..., *ST*_{*N*} be those <simplified term>s; *SPPE* is effectively replaced by:

*PREFIX ST*₁ *SUFFIX* |+| *PREFIX ST*₂ *SUFFIX* |+| ... |+| *PREFIX ST*_{*N*} *SUFFIX*

- iii) If *SC* is a <simplified concatenation> *SCAT*, then let *ST* be the <simplified term> and let *SFL* be the <simplified factor low> simply contained in *SCAT*; *SPPE* is effectively replaced by:

PREFIX ST SUFFIX PREFIX SFL SUFFIX

- iv) If *SC* is a <simplified conjunction> *SAND*, then *SPPE* is effectively replaced by:

EDGEPRE IS SAND EDGESUF

NOTE 255 — As a result, *SAND* is now interpreted as a <label expression> within an <edge pattern>. By earlier Syntax Rules, there are no operators allowed in *SAND* that cannot be interpreted as operators of a <label expression>.

- v) If *SC* is a <simplified quantified> *SQ*, then let *ST* be the <simplified tertiary> simply contained in *SC* and let *GPQ* be the <graph pattern quantifier> simply contained in *SQ*; *SPPE* is effectively replaced by:

(*PREFIX ST SUFFIX*) *GPQ*

- vi) If *SC* is a <simplified questioned>, then let *ST* be the <simplified tertiary> simply contained in *SC*; *SPPE* is effectively replaced by:

(*PREFIX ST SUFFIX*) ?

- vii) If *SC* is a <simplified direction override> *SDO*, then let *SS* be the <simplified secondary> simply contained in *SDO*.

Case:

NOTE 256 — As a result of the following replacements, *SDO* is now interpreted as a <label expression> within an <edge pattern>. By earlier Syntax Rules, there are no operators allowed in *SDO* that cannot be interpreted as operators of a <label expression>.

- 1) If *SDO* is <simplified override left>, then *SPPE* is replaced by:

<- [*IS SS*]-

- 2) If *SDO* is <simplified override undirected>, then *SPPE* is effectively replaced by:

~[*IS SS*]~

- 3) If *SDO* is <simplified override left or undirected>, then *SPPE* is effectively replaced by:

<~[*IS SS*]~

- 4) If *SDO* is <simplified override undirected or right>, then *SPPE* is effectively replaced by:

ISO/IEC 39075:2024(en)
16.12 <simplified path pattern expression>

~[IS SS]~>

- 5) If *SDO* is <simplified override left or right>, then *SPPE* is effectively replaced by:

<- [IS SS] ->

- 6) If *SDO* is <simplified override any direction>, then *SPPE* is effectively replaced by:

-[IS SS]-

- viii) If *SC* is a <simplified negation> *SN*, then *SPPE* is effectively replaced by:

EDGEPRE IS SN EDGESUF

NOTE 257 — As a result, *SN* is now interpreted as a <label expression> within an <edge pattern>. By earlier Syntax Rules, there are no operators allowed in *SN* that cannot be interpreted as operators of a <label expression>.

- ix) If *SC* is a <simplified primary> *SP*, then

Case:

- 1) If *SP* is a <label name>, then *SPPE* is effectively replaced by:

EDGEPRE IS SP EDGESUF

- 2) Otherwise, let *INNER* be the <simplified contents> simply contained in *SC*; *SPPE* is effectively replaced by:

(*PREFIX INNER SUFFIX*)

- 7) The Conformance Rules of [Subclause 16.7](#), “<path pattern expression>” are applied to the result of the previous syntactic transformation.

General Rules

None.

Conformance Rules

- 1) Without Feature G039, “Simplified path pattern expression: full defaulting”, conforming GQL language shall not contain a <simplified path pattern expression> that is not a <simplified defaulting left>, a <simplified defaulting right>, or a <simplified defaulting any direction>.
- 2) Without Feature G080, “Simplified path pattern expression: basic defaulting”, conforming GQL language shall not contain a <simplified defaulting left>, a <simplified defaulting right>, or a <simplified defaulting any direction>.
- 3) Without Feature G081, “Simplified path pattern expression: full overrides”, conforming GQL language shall not contain a <simplified direction override> that is not a <simplified override left>, a <simplified override right>, or a <simplified override any direction>.
- 4) Without Feature G082, “Simplified path pattern expression: basic overrides”, conforming GQL language shall not contain a <simplified override left>, a <simplified override right>, or a <simplified override any direction>.

16.13 <where clause>

Function

Compute a new binding table by selecting records from the current working table fulfilling the specified <search condition>.

Format

```
<where clause> ::=  
  WHERE <search condition>
```

Syntax Rules

- 1) Let *WC* be the <where clause> and let *SC* be the <search condition> immediately contained in *WC*.
- 2) The declared type of the incoming working record of *SC* is the declared type of the incoming working record of *WC* amended with the record type of the declared type of the incoming working table of *WC*.
- 3) The declared type of the incoming working table of *SC* is the material unit binding table type.
- 4) The declared type of *WC* is the declared type of incoming working table of *WC*.

General Rules

- 1) Let *WHERE* be a new empty binding table.
- 2) For each record *R* of the current working table:
 - a) Let *INCLUDE* be the result of *SC* in a new child execution context amended with *R*.
 - b) If *INCLUDE* is *True*, then *R* is added to *WHERE*.
- 3) The result of *WC* is *WHERE*.

Conformance Rules

None.

16.14 <yield clause>

Function

Select and rename columns of a binding table.

Format

```
<yield clause> ::=  
  YIELD <yield item list>
```

```
<yield item list> ::=  
  <yield item> [ { <comma> <yield item> }... ]
```

```
<yield item> ::=  
  { <yield item name> [ <yield item alias> ] }
```

```
<yield item name> ::=  
  <field name>
```

```
<yield item alias> ::=  
  AS <binding variable>
```

Syntax Rules

- 1) Let *YC* be the <yield clause>.
- 2) Let *COYI* be the collection of <yield item>s simply contained in *YC*.
- 3) Let *N* be the number of <yield item>s in *COYI*.
- 4) For each <yield item> *YI_i*, 1 (one) ≤ *i* ≤ *N*, in *COYI*:
 - a) Let *YIN_i* be the <yield item name> specified by *YI_i*.
 - b) If *YI_i* does not immediately contain a <yield item alias>, then:
 - i) *YIN_i* shall be a <binding variable>.
 - ii) *YI_i* is effectively replaced by:
YIN_i AS YIN_i
- 5) Let *YCT* be the <yield clause> after the preceding transformation and let *COYIT* be the collection of *N* <yield item>s simply contained in *YCT*.
- 6) If *COYIT* simply contains a <yield item alias> that simply contains a <binding variable> *YIABV1*, then *COYIT* shall not simply contain another <yield item alias> that simply contains a <binding variable> *YIABV2* such that *YIABV1* and *YIABV2* are equivalent.
- 7) The declared type of *YCT* is a binding table type *BTT* defined as follows:
 - a) *BTT* has *N* columns.
 - b) For each <yield item> *YI_i*, 1 (one) ≤ *i* ≤ *N* in *COYIT*:
 - i) Let *YIN_i* be the <yield item name> specified by *YI_i*.

- ii) Let YIA_i be the <binding variable> simply contained in the <yield item alias> specified by YI_i .
- iii) The declared type of the incoming working table of YC shall have a column SC with column name YIN_i .
- iv) BTT has a column TC .
- v) The column type of TC is the column type of SC .

General Rules

- 1) Let $YIELD$ be a new empty binding table.
- 2) For each record R of the current working table in a new child execution context amended with R :
 - a) Let T be a new empty record.
 - b) For each <yield item> YI_i , $1 \text{ (one)} \leq i \leq N$, from $COYIT$:
 - i) Let YIN_i be the <yield item name> specified by YI_i .
 - ii) Let F_i be the field of the current working record whose name is YIN_i .
 - iii) Let YIV_i be the value of F_i .

NOTE 258 — As opposed to the General Rules for <binding variable>, <yield item>s only consider the current working record and ignore the working records of any parent execution contexts that precede the current execution context in the current execution stack.
 - iv) Let YIA_i be the <binding variable> simply contained in the <yield item alias> specified by YI_i .
 - v) A new field with name YIA_i and with value YIV_i is added to T .
 - c) T is added to $YIELD$.
- 3) The result of YCT is $YIELD$.

Conformance Rules

None.

16.15 <group by clause>

Function

Define a <group by clause> for specifying the set of grouping keys to be used during grouping.

Format

```
<group by clause> ::=  
  GROUP BY <grouping element list>  
  
<grouping element list> ::=  
  <grouping element> [ { <comma> <grouping element> }... ]  
  | <empty grouping set>  
  
<grouping element> ::=  
  <binding variable reference>  
  
<empty grouping set> ::=  
  <left paren> <right paren>
```

Syntax Rules

- 1) Let *GBC* be the <group by clause> and let *GEL* be the <grouping element list>.
- 2) Let *COLS* be the sequence of columns defined as follows.
Case:
 - a) If *GEL* is the <empty grouping set>, then *COLS* is the empty sequence.
 - b) Otherwise:
 - i) Let *GESEQ* be the sequence of <grouping element>s immediately contained in *GEL* and let *NGESEQ* be the number of such <grouping element>s in *GESEQ*.
 - ii) For every *i*-th element *GE_i* of *GESEQ*, $1 \text{ (one)} \leq i \leq \text{NGESEQ}$:
 - 1) Let *BVR_i* be the <binding variable reference> immediately contained in *GE_i*.
 - 2) The declared type of the incoming working record of *BVR_i* is the declared type of the incoming working record of *GBC* amended with the record type of the declared type of the incoming working table of *GBC*.
 - 3) The declared type of the incoming working table of *BVR_i* is the unit binding table type.
 - 4) *BVR_i* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of [Subclause 22.15, "Grouping operations"](#), apply.
 - 5) Let *GEBV_i* be the name of the binding variable referenced by *BVR_i*.
 - 6) Let *COL_i* be the column whose name is *GEBV_i* and whose type is the declared type of *BVR_i*.
 - 7) Append *COL* to *COLS*.

- 3) The declared type of the outgoing working table of *GBC* is a material binding table whose set of columns is *COLS*.
- 4) The declared type of the outgoing working record of *GBC* is the declared type of the incoming working record of *GBC*.
- 5) The declared type of *GBC* is the declared type of the outgoing working table of *GBC*.

General Rules

- 1) Case:
 - a) If *GEL* is the <empty grouping set>, then let *GROUP_BY* be a unit binding table.
 - b) Otherwise:
 - i) For i , $1 \text{ (one)} \leq i \leq \text{NGESEQ}$, let GE_i be the i -th element of *GESEQ* and let $GEBV_i$ be the <binding variable reference> simply contained in $GEBV_i$.
 - ii) Let *GROUP_BY* be a new empty binding table with the column set *COLS*.
 - iii) For each record *R* of the current working table in a new child execution context amended with *R*:
 - 1) Let *T* be a new record comprising fields F_i , $1 \text{ (one)} \leq i \leq \text{NGESEQ}$, such that the name of F_i is the name of $COLS_i$ and the value of F_i is the value of $GEBV_i$ in *R*.
 - 2) If *T* is distinct from every record in *GROUP_BY*, then *T* is added to *GROUP_BY*.
- 2) The result of *GBC* is *GROUP_BY*.

Conformance Rules

- 1) Without Feature GQ15, "GROUP BY clause", conforming GQL language shall not contain <group by clause>.

16.16 <order by clause>

Function

Apply a <sort specification list> to the current working table.

Format

<order by clause> ::=
ORDER BY <sort specification list>

Syntax Rules

- 1) Let *OBC* be the <order by clause>.
- 2) Let *SSL* be the <sort specification list> immediately contained in *OBC*.
- 3) The declared type of *OBC* is the declared type of the incoming working table of *OBC*.

General Rules

- 1) The result of *OBC* is the result of *SSL*.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

16.17 <sort specification list>

Function

Obtaining an ordered binding table from the current working table.

Format

```
<sort specification list> ::=  
  <sort specification> [ { <comma> <sort specification> }... ]  
  
<sort specification> ::=  
  <sort key> [ <ordering specification> ] [ <null ordering> ]  
  
<sort key> ::=  
  <aggregating value expression>  
  
<ordering specification> ::=  
  ASC  
  | ASCENDING  
  | DESC  
  | DESCENDING  
  
<null ordering> ::=  
  NULLS FIRST  
  | NULLS LAST
```

Syntax Rules

- 1) Each <value expression> immediately contained in the <sort key> contained in a <sort specification> is an operand of an ordering operation. The Syntax Rules and Conformance Rules of [Subclause 22.14](#), “Ordering operations”, apply.
- 2) Let SSL be the <sort specification list>
- 3) Let NSS be the number of <sort specification>s immediately contained in SSL .
- 4) For i , $1 \text{ (one)} \leq i \leq NSS$, let SS_i be the i -th <sort specification> immediately contained in SSL .
- 5) For each SS_i , $1 \text{ (one)} \leq i \leq NSS$:
 - a) Let SK_i be the <sort key> immediately contained in SS_i .
 - b) If SS_i does not immediately contain an <ordering specification>, then SS_i is effectively replaced by:
 $SK_i \text{ ASC}$
 - c) The declared type of the incoming working record of SK_i is the declared type of the incoming working record of SSL amended with the record type of the declared type of the incoming working table of SSL .
 - d) The declared type of the incoming working table of SK_i is the material unit binding table type.
 - e) The declared type of SK_i is the declared type of the <aggregating value expression> immediately contained in SK_i .

- 6) If <null ordering> is not specified, then an implementation-defined (IS001) <null ordering> is implicit. The implementation-defined default for <null ordering> shall not depend on the context outside of <sort specification list>.
- 7) The declared type of *SSL* is the declared type of the incoming working table of *SSL*.

General Rules

- 1) Let *SORTED* be a new ordered binding table created from the collection of records of the current working table by ordering the records, as follows:
 - a) Let *N* be the number of <sort specification>s.
 - b) For *i*, $1 \text{ (one)} \leq i \leq N$, let K_i be the <sort key> contained in the *i*-th <sort specification>.
 - c) Each <sort specification> specifies the *sort direction* for the corresponding sort key K_i . If neither DESC nor DESCENDING is specified in the *i*-th <sort specification>, then the sort direction for K_i is ascending and the applicable <comp op> is the <less than operator>; otherwise, the sort direction for K_i is descending and the applicable <comp op> is the <greater than operator>.
 - d) Let *P* be any record of the collection of records to be ordered, and let *Q* be any other record of the same collection of records.
 - e) Let PV_i be the result of K_i in a new child execution context amended with *P*.
 - f) Let QV_i be the result of K_i in a new child execution context amended with *Q*.
 - g) The relative position of records *P* and *Q* in the result is determined by comparing PV_i and QV_i as follows:
 - i) The comparison is performed according to the General Rules of Subclause 19.3, "<comparison predicate>", where the <comp op> is the applicable <comp op> for K_i .
 - ii) The comparison is performed with the following special treatment of null values.

Case:

 - 1) If PV_i and QV_i are both the null value, then they are considered equal to each other.
 - 2) If PV_i is the null value and QV_i is not the null value, then

Case:

 - A) If NULLS FIRST is specified or implied, then $PV_i < \text{comp op} > QV_i$ is considered to be True.
 - B) If NULLS LAST is specified or implied, then $PV_i < \text{comp op} > QV_i$ is considered to be False.
 - 3) If PV_i is not the null value and QV_i is the null value, then

Case:

 - A) If NULLS FIRST is specified or implied, then $PV_i < \text{comp op} > QV_i$ is considered to be False.
 - B) If NULLS LAST is specified or implied, then $PV_i < \text{comp op} > QV_i$ is considered to be True.

- h) PV_i is said to *precede* QV_i if the result of the <comparison predicate> " PV_i <comp op> QV_i " is True for the applicable <comp op>.
- i) If PV_i and QV_i are not the null value and the result of " PV_i <comp op> QV_i " is Unknown, then the relative ordering of PV_i and QV_i is implementation-dependent (US007).
- j) The relative position of record P is before record Q if for some n , $1 \text{ (one)} \leq n \leq N$, PV_n precedes QV_n and PV_i is not distinct from QV_i for all $i < n$.
- k) Two records that are not distinct with respect to the <sort specification>s are said to be *peers* of each other. The relative ordering of peers is implementation-dependent (US006).
- l) The result of *SSL* is *SORTED*.

Conformance Rules

- 1) Without Feature GA03, "Explicit ordering of nulls", conforming GQL language shall not contain a <null ordering>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

16.18 <limit clause>

Function

Obtain a new binding table that retains only a limited number of records of the current working table.

Format

<limit clause> ::=
LIMIT <non-negative integer specification>

Syntax Rules

- 1) Let *LC* be the <limit clause>.
- 2) The declared type of *LC* is the declared type of the incoming working table of *LC*.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) If *TABLE* is not ordered, then let *ORDERED_TABLE* be a new ordered binding table created from the result of sorting the collection of all records of *TABLE* according to an implementation-dependent (US001) order; otherwise, let *ORDERED_TABLE* be *TABLE*.
- 3) Let *V* be the result of the <non-negative integer specification>.
- 4) Let *LIMIT* be a new ordered binding table obtained by selecting only the first *V* records of *ORDERED_TABLE* and discarding all subsequent records.

NOTE 259 — If the order of the result of *LC* has no effect on the outcome of a <GQL-program>, there is no need for *LIMIT* to be indicated as ordered.

- 5) The result of *LC* is *LIMIT*.

Conformance Rules

None.

16.19 <offset clause>

Function

Obtaining a new binding table that retains all records of the current working table except for some discarded initial records.

Format

```
<offset clause> ::=  
  <offset synonym> <non-negative integer specification>  
  
<offset synonym> ::=  
  OFFSET | SKIP
```

Syntax Rules

- 1) Let *OC* be the <offset clause>.
- 2) The declared type of *OC* is the declared type of the incoming working table of *OC*.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) If *TABLE* is not ordered, then let *ORDERED_TABLE* be a new ordered binding table created from the result of sorting the collection of all records of *TABLE* according to an implementation-dependent (US001) order; otherwise, let *ORDERED_TABLE* be *TABLE*.

NOTE 260 — If the order of the result of *OC* has no effect on the outcome of a <GQL-program>, there is no need for *OFFSET* to be indicated as ordered.
- 3) Let *V* be the result of the <non-negative integer specification>.
- 4) Let *OFFSET* be a new ordered binding table obtained from all but the first *V* records of *ORDERED_TABLE*.
- 5) The result of *OC* is *OFFSET*.

Conformance Rules

None.

17 Object references

17.1 <schema reference> and <catalog schema parent and name>

Function

Identify a GQL-schema in the GQL-catalog.

Format

```

<schema reference> ::=
    <absolute catalog schema reference>
  | <relative catalog schema reference>
  | <reference parameter specification>

<absolute catalog schema reference> ::=
    <solidus> | <absolute directory path> <schema name>

<catalog schema parent and name> ::=
    <absolute directory path> <schema name>

<relative catalog schema reference> ::=
    <predefined schema reference>
  | <relative directory path> <schema name>

<predefined schema reference> ::=
    HOME_SCHEMA | CURRENT_SCHEMA | <period>

<absolute directory path> ::=
    <solidus> [ <simple directory path> ]

<relative directory path> ::=
    <double period>
    [ { <solidus> <double period> }... <solidus> [ <simple directory path> ] ]

<simple directory path> ::=
    { <directory name> <solidus> }...

```

Syntax Rules

- 1) If the <schema reference> *SR* is specified, then the GQL-schema identified by *SR* is the GQL-schema identified by the immediately contained <absolute catalog schema reference> or <relative catalog schema reference>.
- 2) If the <absolute catalog schema reference> *ACSR* is specified, then

Case:

 - a) If *ACSR* is a <solidus>, then:
 - i) The GQL-catalog root shall be a GQL-schema.
 - ii) The GQL-schema identified by *ACSR* is the GQL-catalog root.
 - b) Otherwise:

17.1 <schema reference> and <catalog schema parent and name>

- i) Let *PARENT* be the GQL-directory identified by the immediately contained <absolute directory path>.
 - ii) Let *SN* be the immediately contained <schema name>.
 - iii) *PARENT* shall contain a GQL-schema *S* with name *SN*.
 - iv) For every GQL-directory or GQL-schema *DOS* contained in *PARENT*, if the name by which *DOS* is identified in *PARENT* and *SN* are visually confusable with each other, then the following exception condition is raised: *syntax error or access rule violation — use of visually confusable identifiers (42004)*.
 - v) The GQL-schema identified by *ACSR* is *S*.
- 3) If the <relative catalog schema reference> *RCSR* is specified, then the GQL-schema identified by *RCSR* is defined as follows.
- Case:
- a) If *RCSR* is the <predefined schema reference> *PSR*, then the GQL-schema identified by *RCSR* is the GQL-schema identified by *PSR*.
 - b) If *RCSR* immediately contains the <relative directory path> *RDP* and the <schema name> *SN*, then:
 - i) Let *PARENT* be the GQL-directory identified by *RDP*.
 - ii) *SN* shall identify an existing GQL-schema descriptor in *PARENT*.
 - iii) The schema identified by *RCSR* is the GQL-schema identified by *SN* in *PARENT*.
- 4) If the <predefined schema reference> *PSR* is specified, then the GQL-schema identified by *PSR* is defined as follows.
- Case:
- a) If *PSR* is HOME_SCHEMA, then:
 - i) The current home schema shall not be “omitted”.
 - ii) *PSR* identifies the current home schema.
 - b) If *PSR* is CURRENT_SCHEMA or <period>, then:
 - i) The current working schema of *PSR* shall not be “omitted”.
 - ii) *PSR* identifies the current working schema of *PSR*.
- 5) If the <absolute directory path> *ADP* is specified, then:
- a) The GQL-catalog root shall be a GQL-directory.
 - b) The GQL-directory identified by *ADP* is defined as follows.
- Case:
- i) If *ADP* does not immediately contain the <simple directory path>, then the GQL-directory identified by *ADP* is the GQL-catalog root.
 - ii) Otherwise:
 - 1) Let *SDP* be the <simple directory path> immediately contained in *ADP*. The Syntax Rules of Subclause 22.9, “Resolution of a <simple directory path> from a start directory”, are applied with *SDP* as *SIMPLE DIRECTORY PATH* and the GQL-catalog

17.1 <schema reference> and <catalog schema parent and name>

root as *START DIRECTORY*; let *RD* be the *RESOLVED DIRECTORY* returned from the application of those Syntax Rules.

2) The GQL-directory identified by *ADP* is *RD*.

6) If the <relative directory path> *RDP* is specified, then:

a) Let *N* be the number of all <double period>s that are immediately contained in *RDP*.

b) The current working schema of *RDP* shall not be “omitted”.

c) Let *PD_{N+1}* be the current working schema of *RDP*.

d) For *j*, $N \geq j \geq 1$ (one), the GQL-directory *PD_j* is defined as follows:

i) GQL-directory *PD_{j+1}* shall not be the GQL-catalog root.

ii) *PD_j* is the parent directory of the GQL-directory or GQL-schema *PD_{j+1}*.

e) The GQL-directory identified by *RDP* is defined as follows.

Case:

i) If *RDP* does not immediately contain the <simple directory path>, then the GQL-directory identified by *RDP* is the GQL-directory *PD₁*.

ii) Otherwise:

1) Let *SDP* be the <simple directory path> immediately contained in *RDP*. The Syntax Rules of Subclause 22.9, “Resolution of a <simple directory path> from a start directory”, are applied with *RDP* as *SIMPLE DIRECTORY PATH* and *PD₁* as *START DIRECTORY*; let *RD* be the *RESOLVED DIRECTORY* returned from the application of those Syntax Rules.

2) The GQL-directory identified by *RDP* is *RD*.

General Rules

None.

Conformance Rules

None.

17.2 <graph reference> and <catalog graph parent and name>

Function

Identify a graph in the GQL-catalog.

Format

```
<graph reference> ::=  
  <catalog object parent reference> <graph name>  
  | <delimited graph name>  
  | <home graph>  
  | <reference parameter specification>  
  
<catalog graph parent and name> ::=  
  [ <catalog object parent reference> ] <graph name>  
  
<home graph> ::=  
  HOME_PROPERTY_GRAPH | HOME_GRAPH
```

Syntax Rules

- 1) If <graph reference> *GR* is specified, then
Case:
 - a) If *GR* is a <home graph>, then:
 - i) The current home graph shall not be “omitted”.
 - ii) *GR* identifies the current home graph.
 - b) If *GR* is a <delimited graph name> *DGN*, then the graph identified by *GR* is the graph identified by the <catalog graph parent and name>:
./DGN
 - c) Otherwise, *GR* simply contains a <catalog object parent reference> *COPR* and a <graph name> *GN* and the graph identified by *GR* is the graph identified by the <catalog graph parent and name>:
COPR GN
- 2) If <catalog graph parent and name> *CGPN* is specified, then:
 - a) If *CGPN* does not immediately contain a <catalog object parent reference>, then the following <catalog object parent reference> is implicit:
./
 - b) Let *PARENT* be the GQL-schema or the catalog object identified by the explicit or implicit <catalog object parent reference> immediately contained in *CGPN*.
 - c) Let *GN* be the <graph name> immediately contained in *CGPN*.
 - d) Case:
 - i) If *PARENT* contains an object *O* with name *GN*, then:

17.2 <graph reference> and <catalog graph parent and name>

- 1) *O* shall be a graph.
 - 2) For every primary object *PO* contained in *PARENT*, if the name by which *PO* is identified in *PARENT* and *GN* are visually confusable with each other, then the following exception condition is raised: *syntax error or access rule violation — use of visually confusable identifiers (42004)*.
 - 3) The graph identified by *CGPN* is *O*.
- ii) Otherwise, *CGPN* does not identify any object.

General Rules

None.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

17.3 <graph type reference> and <catalog graph type parent and name>

Function

Identify a graph type in the GQL-catalog.

Format

```
<graph type reference> ::=
  <catalog graph type parent and name>
  | <reference parameter specification>

<catalog graph type parent and name> ::=
  [ <catalog object parent reference> ] <graph type name>
```

Syntax Rules

- 1) If <graph type reference> *GTR* is specified, then the immediately contained <catalog graph type parent and name> *CGTPN* shall identify a graph type *GT*. The graph type identified by *GTR* is *GT*.
- 2) If the <catalog graph type parent and name> *CGTPN* is specified, then:
 - a) If *CGTPN* does not immediately contain a <catalog object parent reference>, then the following <catalog object parent reference> is implicit:

./
 - b) Let *PARENT* be the GQL-schema or the catalog object identified by the explicit or implicit <catalog object parent reference> immediately contained in *CGTPN*.
 - c) Let *GTN* be the <graph type name> immediately contained in *CGTPN*.
 - d) Case:
 - i) If *PARENT* contains an object *O* with name *GTN*, then:
 - 1) *O* shall be a graph type.
 - 2) For every primary object *PO* contained in *PARENT*, if the name by which *PO* is identified in *PARENT* and *GTN* are visually confusable with each other, then the following exception condition is raised: *syntax error or access rule violation — use of visually confusable identifiers (42004)*.
 - 3) The graph type identified by *CGTPN* is *O*.
 - ii) Otherwise, *CGTPN* does not identify any object.

General Rules

None.

Conformance Rules

None.

17.4 <binding table reference> and <catalog binding table parent and name>

Function

Identify a binding table in the GQL-catalog.

Format

```
<binding table reference> ::=
  <catalog object parent reference> <binding table name>
  | <delimited binding table name>
  | <reference parameter specification>

<catalog binding table parent and name> ::=
  [ <catalog object parent reference> ] <binding table name>
```

Syntax Rules

- 1) If the <binding table reference> *BTR* is specified, then

Case:

 - a) If *BTR* is a <delimited binding table name> *DBTN*, then the binding table identified by *BTR* is the binding table identified by the <catalog binding table parent and name>:

./DBTN
 - b) Otherwise, *BTR* simply contains a <catalog object parent reference> *COPR* and a <binding table name> *BTN* and the binding table identified by *BTR* is the binding table identified by the <catalog binding table parent and name>:

COPR BTN
- 2) If the <catalog binding table parent and name> *CBTPN* is specified, then:
 - a) If the <catalog binding table parent and name> does not immediately contain a <catalog object parent reference>, then the following <catalog object parent reference> is implicit:

./
 - b) Let *PARENT* be the GQL-schema or the catalog object identified by the <catalog object parent reference> immediately contained in *CBTPN*.
 - c) Let *BTN* be the <binding table name> immediately contained in *CBTPN*.
 - d) Case:
 - i) If *PARENT* contains an object *O* with name *BTN*, then:
 - 1) *O* shall be a binding table.
 - 2) For every primary object *PO* contained in *PARENT*, if the name by which *PO* is identified in *PARENT* and *BTN* are visually confusable with each other, then the following exception condition is raised: *syntax error or access rule violation — use of visually confusable identifiers (42004)*.
 - 3) The binding table identified by *CBTPN* is *O*.

17.4 <binding table reference> and <catalog binding table parent and name>

- ii) Otherwise, *CBTPN* does not identify any object.

General Rules

None.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

17.5 <procedure reference> and <catalog procedure parent and name>

Function

Identify a procedure in the GQL-catalog.

Format

```
<procedure reference> ::=  
    <catalog procedure parent and name>  
    | <reference parameter specification>  
  
<catalog procedure parent and name> ::=  
    [ <catalog object parent reference> ] <procedure name>
```

Syntax Rules

- 1) If the <catalog procedure parent and name> does not immediately contain a <catalog object parent reference>, then the following <catalog object parent reference> is implicit:
./
- 2) If the <procedure reference> *PR* is specified, then the procedure identified by *PR* is the procedure identified by the immediately contained <catalog procedure parent and name>.
- 3) If the <catalog procedure parent and name> *CPPN* is specified, then:
 - a) Let *PARENT* be the GQL-schema or the catalog object identified by the <catalog object parent reference> immediately contained in *CPPN*.
 - b) The procedure identified by *CPPN* is defined as follows:
 - i) Let *PN* be the <procedure name> immediately contained in *CPPN*.
 - ii) *PARENT* shall contain a procedure *P* with name *PN*.
 - iii) For every primary object *PO* contained in *PARENT*, if the name by which *PO* is identified in *PARENT* and *PN* are visually confusable with each other, then the following exception condition is raised: *syntax error or access rule violation — use of visually confusable identifiers (42004)*.
 - iv) The procedure identified by *CPPN* is *P*.

General Rules

None.

Conformance Rules

None.

17.6 <catalog object parent reference>

Function

Identify the parent GQL-schema or parent catalog object in a reference to a catalog object.

Format

```
<catalog object parent reference> ::=  
  <schema reference> [ <solidus> ] [ { <object name> <period> }... ]  
  | { <object name> <period> }...
```

Syntax Rules

- 1) Let *COPR* be the <catalog object parent reference>.
Case:
 - a) If *COPR* simply contains a <schema reference> that is a <solidus>, then *COPR* shall not immediately contain a <solidus>.
 - b) Otherwise, *COPR* shall immediately contain a <solidus>.NOTE 261 — // is not a valid <catalog object parent reference>.
- 2) Let the GQL-schema *SCHEMA* be defined as follows.
Case:
 - a) If *COPR* immediately contains a <schema reference> *CSR*, then *SCHEMA* is the GQL-schema identified by *CSR*.
 - b) Otherwise:
 - i) The current working schema of *COPR* shall not be “omitted”.
 - ii) *SCHEMA* is the current working schema of *COPR*.
- 3) Let *ONSEQ* be the sequence of all <object name>s that are immediately contained in *COPR*. Let *N* be the number of elements of *ONSEQ*.
- 4) For *i*, $1 \text{ (one)} \leq i \leq N$, let ON_i be the *i*-th element of *ONSEQ*.
- 5) The GQL-schema or catalog object identified by *COPR* is defined as follows.
Case:
 - a) If *N* is 0 (zero), then *COPR* identifies the GQL-schema *SCHEMA*.
 - b) Otherwise,
 - i) The GQL-schema *SCHEMA* shall contain a catalog object with name ON_1 .
 - ii) Let CO_1 be the catalog object with name ON_1 contained in *SCHEMA*.
 - iii) For *j*, $2 \text{ (two)} \leq j \leq N$, the catalog object CO_j is defined as follows:
 - 1) The catalog object CO_{j-1} shall contain a catalog object with name ON_j .

- 2) CO_j is the catalog object with name ON_j contained in CO_{j-1} .
- 6) $COPR$ identifies the catalog object CO_N .

General Rules

None.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

17.7 <reference parameter specification>

Function

Specify a reference parameter for a catalog reference.

Format

<reference parameter specification> ::=
 <substituted parameter reference>

Syntax Rules

1) Let *RPS* be the <reference parameter specification>.

2) Let *SPR* be the <substituted parameter reference> simply contained in *RPS*.

3) Let *ROP* be the parameter substitution of *SPR*.

NOTE 262 — The parameter substitution of a <substituted parameter reference> is determined by the provisions of Subclause 22.1, “Annotation of a <GQL-program>”.

4) Let *ROP1* be defined as follows.

Case:

a) If *RPS* is immediately contained in a <schema reference>, then:

i) One of the following shall hold:

- 1) *ROP* conforms to the Format and Syntax Rules of <absolute catalog schema reference>.
- 2) *ROP* conforms to the Format and Syntax Rules of <relative catalog schema reference>.

ii) *ROP1* is *ROP*.

b) If *RPS* is immediately contained in a <graph reference>, then:

i) One of the following shall hold:

- 1) *ROP* conform to the Format and Syntax Rules of <catalog graph parent and name>.
- 2) *ROP* conforms to the Format and Syntax Rules of <home graph>.

ii) The Syntax Rules of Subclause Subclause 17.2, “<graph reference> and <catalog graph parent and name>” are applied to *ROP*; *ROP1* is *ROP* after the application of those Syntax Rules.

c) If *RPS* is immediately contained in a <graph type reference>, then *ROP* shall conform to the Format and Syntax Rules of <catalog graph type parent and name> and *ROP1* is *ROP*.

d) If *RPS* is immediately contained in a <binding table reference>, then:

i) *ROP* shall conform to the Format and Syntax Rules of <catalog binding table parent and name>.

ISO/IEC 39075:2024(en)
17.7 <reference parameter specification>

- ii) The Syntax Rules of Subclause [Subclause 17.4, “<binding table reference> and <catalog binding table parent and name>”](#) are applied to *ROP*; *ROP1* is *ROP* after the application of those Syntax Rules.
- e) If *RPS* is immediately contained in a <procedure reference>, then *ROP* shall conform to the Format and Syntax Rules of <catalog procedure parent and name> and *ROP1* is *ROP*.
- 5) *RPS* is effectively replaced by *ROP1*.

General Rules

None.

Conformance Rules

- 1) Without Feature GE08, “Reference parameters”, conforming GQL language shall not contain a <reference parameter specification>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

17.8 <external object reference>

Function

Identify a GQL-object with a URI.

Format

<external object reference> ::=
!! See the Syntax Rules.

Syntax Rules

- 1) Let *EOR* be the <external object reference>.
- 2) *EOR* shall be a URI with a mandatory scheme as specified by RFC 3986 or alternatively shall be an absolute-URL-with-fragment character string as specified by WHATWG URL.
- 3) *EOR* shall contain a <colon>.

NOTE 263 — This condition is true for any URI with a mandatory scheme as specified by RFC 3986 and, when this document was last edited, was true for any absolute-URL-with-fragment character string as specified by WHATWG URL.
- 4) It is implementation-defined (IE001) what *EOR* identifies.
- 5) *EOR* shall identify a GQL-object.

General Rules

None.

Conformance Rules

- 1) Without Feature GH01, “External object references”, conforming GQL language shall not contain an <external object reference>.

18 Type elements

18.1 <nested graph type specification>

Function

Specify a graph type.

Format

```
<nested graph type specification> ::=
  <left brace> <graph type specification body> <right brace>

<graph type specification body> ::=
  <element type list>

<element type list> ::=
  <element type specification> [ { <comma> <element type specification> }... ]

<element type specification> ::=
  <node type specification>
  | <edge type specification>
```

Syntax Rules

- 1) Let *NGTS* be the <nested graph type specification>.
- 2) Let *GTSB* be the <graph type specification body> simply contained in *NGTS*.
- 3) An *element type specification* of *GTSB* is a <node type specification> or an <edge type specification> simply contained in *GTSB*.
- 4) The *property names* of an element type specification *AETS* are given by the set of names of all property types of *AETS*.
- 5) For two element type specifications *ET1* and *ET2* of a <graph type specification body>, whether *ET1* and *ET2* are *property name-sharing* is defined as follows:
Case:
 - a) If the intersection of the property names of *ET1* and *ET2* is non-empty, then *ET1* and *ET2* are property name-sharing.
 - b) Otherwise, the intersection of property names of *ET1* and *ET2* is empty and *ET1* and *ET2* are not property name-sharing.
- 6) For an element type specification *SUB_ELTS* and an element type specification *SUPER_ELTS*, whether *SUB_ELTS* *implies* *SUPER_ELTS* is defined as follows.
Case:
 - a) If all the following are true, then *SUB_ELTS* *implies* *SUPER_ELTS*.
 - i) Either *SUB_ELTS* and *SUPER_ELTS* both are node types or they both are edge types.

ISO/IEC 39075:2024(en)
18.1 <nested graph type specification>

- ii) The effective key label set of *SUPER_ELTS* is not “omitted”.
 - iii) The effective key label set of *SUPER_ELTS* is a subset of the label set of *SUB_ELTS*.
 - iv) One of the following is true:
 - 1) The effective key label set of *SUB_ELTS* is “omitted”.
 - 2) The effective key label set of *SUPER_ELTS* and the effective key label set of *SUB_ELTS* are different.
 - b) Otherwise, *SUB_ELTS* does not imply *SUPER_ELTS*.
- 7) For an element type specification *SUB_ELTS* and an element type specification *SUPER_ELTS*, whether *SUB_ELTS* is *structurally consistent* with *SUPER_ELTS* is defined as follows.
- Case:
- a) If all the following are true, then *SUB_ELTS* is structurally consistent with *SUPER_ELTS*:
 - i) Either *SUB_ELTS* and *SUPER_ELTS* both are node types or they both are edge types.
 - ii) The label set of *SUPER_ELTS* is a subset of the label set of *SUB_ELTS*.
 - iii) The property names of *SUPER_ELTS* is a subset of the property names of *SUB_ELTS*.
 - iv) For every property name *PN* of *SUPER_ELTS*, all of the following are true:
 - 1) Let *VT_SUPER* be the value type of the property type with name *PN* in the property type set of *SUPER_ELTS*.
 - 2) Let *VT_SUB* be the value type of the property type with name *PN* in the property type set of *SUB_ELTS*.
 - 3) A hypothetical application of the Syntax Rules of Subclause 22.17, “Graph-type specific combination of property value types”, with the set comprising *VT_SUPER* and *VT_SUB* as *DTSET* would succeed; let *CVT* be the *RESTYPE* that would be returned from such a hypothetical application.
 - 4) *CVT* is *VT_SUPER*.
 - b) Otherwise, *SUB_ELTS* is not structurally consistent with *SUPER_ELTS*.
- 8) For every <node type specification> *NTS* simply contained in *GTSB* and whose effective key label set is not “omitted”, *GTSB* shall not simply contain a <node type specification> *ONTS* other than *NTS* such that the effective key label set of *NTS* and the effective key label set of *ONTS* are the same.
- 9) For every <node type specification> *ONTS* that is simply contained in *GTSB* and that implies *NTS*, *ONTS* shall be structurally consistent with *NTS*.
- 10) For every <node type specification> *NTS* simply contained in *GTSB*, *GTSB* shall not simply contain a <node type specification> *ONTS* other than *NTS* such that the local node type alias of *NTS* and the local node type alias of *ONTS* are the same.
- 11) For every <edge type specification> *ETS* simply contained in *GTSB*:
- a) *GTSB* shall simply contain a <node type specification> that specifies the node type that is the source node type of *ETS*.
 - b) *GTSB* shall simply contain a <node type specification> that specifies the node type that is the destination node type of *ETS*.

ISO/IEC 39075:2024(en)
18.1 <nested graph type specification>

- 12) The *source node type specification* of an <edge type specification> *AETS* simply contained in *GTSB* is the <node type specification> simply contained in *GTSB* that specifies the node type that is the same type as the source node type of *AETS*.
- 13) The *destination node type specification* of an <edge type specification> *AETS* simply contained in *GTSB* is the <node type specification> simply contained in *GTSB* that specifies the node type that is the same type as the destination node type of *AETS*.
- 14) For every <edge type specification> *ETS* that is simply contained in *GTSB* and whose effective key label set is not “omitted”, *GTSB* shall not simply contain an <edge type specification> *OETS* other than *ETS* such that the effective key label set of *ETS* and the effective key label set of *OETS* are the same and at least one of the following is true:
 - a) *ETS* specifies a directed edge type and *OETS* specifies an undirected edge type.
 - b) *ETS* specifies an undirected edge type and *OETS* specifies a directed edge type.
 - c) The label set of *ETS* and the label set of *OETS* are not the same.
 - d) The property type set of *ETS* and the property type set of *OETS* are not the same.
- 15) For an <edge type specification> *SUB_ETTS* implying an <edge type specification> *SUPER_ETTS*, whether *SUB_ETTS* is *structurally endpoint-consistent* with *SUPER_ETTS* is defined as follows.

Case:

 - a) If one of the following is true, then *SUB_ETTS* is *structurally endpoint-consistent* with *SUPER_ETTS*:
 - i) *SUPER_ETTS* and *SUB_ETTS* both specify directed edge types, the source node type specification of *SUB_ETTS* is structurally consistent with the source node type specification of *SUPER_ETTS* and the destination node type specification of *SUB_ETTS* is structurally consistent with the destination node type specification of *SUPER_ETTS*.
 - ii) *SUPER_ETTS* and *SUB_ETTS* both specify undirected edge types and one of the following is true:
 - 1) The source node type specification of *SUB_ETTS* is structurally consistent with the source node type specification of *SUPER_ETTS* and the destination node type specification of *SUB_ETTS* is structurally consistent with the destination node type specification of *SUPER_ETTS*.
 - 2) The source node type specification of *SUB_ETTS* is structurally consistent with the destination node type specification of *SUPER_ETTS* and the destination node type specification of *SUB_ETTS* is structurally consistent with the source node type specification of *SUPER_ETTS*.
 - b) Otherwise, *SUB_ETTS* is not *structurally endpoint-consistent* with *SUPER_ETTS*.
- 16) For every <edge type specification> *OETS* that is simply contained in *GTSB* and implies *ETS*, all of the following shall be true:
 - a) *OETS* is structurally consistent with *ETS*.
 - b) *OETS* is structurally endpoint-consistent with *ETS*.
- 17) The *set of node type key label sets* of a <graph type specification body> *AGTSB* is the set of the effective key label sets of all <node type specification>s simply contained in *AGTSB* except for those <node type specification>s whose effective key label sets are “omitted”.
- 18) The *set of edge type key label sets* of a <graph type specification body> *AGTSB* is the set of the effective key label sets of all <edge type specification>s simply contained in *AGTSB* except for those <edge type specification>s whose effective key label sets are “omitted”.

- 19) *NGTS* specifies the graph type specified by *GTSB*.
- 20) *GTSB* specifies the graph type with
 - a) The node type set that is the set of node types specified by the <node type specification>s simply contained in *GTSB*.
 - b) The edge type set that is the set of edge types specified by the <edge type specification>s simply contained in *GTSB*.
- 21) The node type key label set dictionary of a graph type *AGT* specified by a <graph type specification body> *AGTSB* is a dictionary that maps every key label set *NTKLS* in the set of node type key label sets of *AGTSB* to the node type specified by a <node type specification> *NTS* that is simply contained in *AGTSB* and where *NTKLS* is the key label set of *NTS*.
- 22) The edge type key label set dictionary of a graph type *AGT* specified by a <graph type specification body> *AGTSB* is a dictionary that maps every key label set *ETKLS* in the set of edge type key label sets of *AGTSB* to the set of all edge types specified by an <edge type specification> *ETS* that is simply contained in *AGTSB* and where *ETKLS* is the key label set of *ETS*.
- 23) For each graph type *GT*, there is an implementation-defined (IV003) graph type, *GTNF(GT)*, known as the normal form of *GT* (which can be *GT* itself), such that:
 - a) If *GT1* and *GT2* are two graph types, the sets of the normal forms of the node types of *GT1* and *GT2* are the same, and the sets of the normal forms of the edge types of *GT1* and *GT2* are the same, then $GTNF(GT1) = GTNF(GT2)$.
 - b) $GTNF(GTNF(GT)) = GTNF(GT)$.
 - c) The results of all invocations of $GTNF(GT)$ are the same.

NOTE 265 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a graph type from amongst the equivalent graph types are deterministic.

General Rules

- 1) The graph type descriptor is created for the graph type *GT* specified by *GTSB* that describes *GT* and comprises:
 - a) The declared name of the primary base type of all graph types (GRAPH DATA).

NOTE 266 — See Subclause 4.13.2.2, “Graph type descriptors”.
 - b) The preferred name of *GT*.
 - c) The set of the node type descriptors of all node types of *GT*.
 - d) The set of the edge type descriptors of all edge types of *GT*.
 - e) The node type key label set dictionary of *GT*.
 - f) The edge type key label set dictionary of *GT*.

Conformance Rules

- 1) Without Feature GG23, “Optional element type key label sets”, in conforming GQL language, a <graph type specification body> shall not contain a <node type specification> or an <edge type specification> whose effective key label set is “omitted”.
- 2) Without Feature GG24, “Relaxed structural consistency”, in conforming GQL language, for every two property name-sharing element type specifications *ET1* and *ET2* of a <graph type specification

ISO/IEC 39075:2024(en)
18.1 <nested graph type specification>

body> and every property name *PN* in the intersection of the property names of *ET1* and *ET2*, it holds that a hypothetical application of the Syntax Rules of [Subclause 22.17](#), “Graph-type specific combination of property value types” with the set comprising the value types of the property types of *ET1* and *ET2* whose name is *PN* as *DTSET* would succeed.

- 3) Without Feature GG25, “Relaxed key label set uniqueness for edge types”, in conforming GQL language, a <graph type specification body> shall not simply contain two <edge type specification>s whose effective key label sets both are the same but are not “omitted”.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

18.2 <node type specification>

Function

Specify a node type.

Format

```
<node type specification> ::=  
  <node type pattern>  
  | <node type phrase>  
  
<node type pattern> ::=  
  [ <node synonym> [ TYPE ] <node type name> ]  
  <left paren> [ <local node type alias> ] [ <node type filler> ] <right paren>  
  
<node type phrase> ::=  
  <node synonym> [ TYPE ] <node type phrase filler>  
  [ AS <local node type alias> ]  
  
<node type phrase filler> ::=  
  <node type name> [ <node type filler> ]  
  | <node type filler>  
  
<node type filler> ::=  
  <node type key label set> [ <node type implied content> ]  
  | <node type implied content>  
  
<local node type alias> ::=  
  <regular identifier>  
  
<node type implied content> ::=  
  <node type label set>  
  | <node type property types>  
  | <node type label set> <node type property types>  
  
<node type key label set> ::=  
  [ <label set phrase> ] <implies>  
  
<node type label set> ::=  
  <label set phrase>  
  
<node type property types> ::=  
  <property types specification>
```

Syntax Rules

- 1) Let *NTS* be the <node type specification>.
- 2) If *NTS* is simply contained in a <closed node reference value type>, then *NTS* shall not simply contain a <local node type alias>, a <node type key label set>, or a <node type name>.
- 3) If *NTS* simply contains a <node type name>, then:
 - a) *NTS* shall not simply contain a <node type key label set>.
 - b) *NTS* shall not simply contain a <local node type alias>.
- 4) The *local node type alias* of a <node type specification> *ANTS* is defined as follows.

ISO/IEC 39075:2024(en)
18.2 <node type specification>

Case:

- a) If *ANTS* simply contains an implicit or explicit <local node type alias> *LNTA*, then the local node type alias of *ANTS* is the name specified by the <regular identifier> that constitutes *LNTA*.
- b) Otherwise, no <local node type alias> is suitably specified and the local node type alias of *ANTS* is the name of a new system-generated regular identifier.

- 5) The *key label set* of a <node type specification> *ANTS* is defined as follows.

Case:

- a) If *ANTS* simply contains a <node type key label set> *NTKLS* that simply contains a <label set phrase> *LSP*, then the key label set of *ANTS* is the label set specified by *LSP*.
- b) If *ANTS* simply contains a <node type key label set> *NTKLS* that does not simply contain a <label set phrase>, then the key label set of *ANTS* is an empty label set.
- c) If *ANTS* simply contains a <node type name> *NTN*, then the key label set of *ANTS* is the label set specified by the following <label set phrase>:

: *NTN*

- d) Otherwise, no key label set is suitably specified and the key label set of *ANTS* is “omitted”.

- 6) The *implied label set* of a <node type specification> *ANTS* is defined as follows.

Case:

- a) If *ANTS* simply contains an implicit or explicit <node type label set> *NTLS*, then the implied label set of *ANTS* is the label set specified by the <label set phrase> that constitutes *NTLS*.
- b) Otherwise, no <node type label set> is suitably specified and the implied label set of *ANTS* is the empty label set.

- 7) If the key label set of *NTS* is not “omitted”, then the key label set of *NTS* and the implied label set of *NTS* shall be disjoint.

- 8) The *label set* of a <node type specification> *ANTS* is the union of the key label set of *ANTS* and the implied label set of *ANTS*.

- 9) The *effective key label set* of a <node type specification> *ANTS* is defined as follows.

Case:

- a) If *ANTS* is simply contained in a <graph type specification body> *GTSB*, then

Case:

- i) If Feature GG22, “Element type key label set inference” is supported and the key label set of *ANTS* is “omitted”, then:

- 1) Let *ONTSS* be the set of all <node type specification>s simply contained in *GTSB* other than *ANTS*.
- 2) Let *OLS* be the union of the label sets of all <node type specification>s in *ONTSS*.
- 3) Let *ULS* be the set of all labels that are in the implied label set of *ANTS* but not in *OLS*.
- 4) If *ULS* is an empty label set, then the effective key label set of *ANTS* is “omitted”; otherwise, the effective key label set of *ANTS* is *ULS*.

ISO/IEC 39075:2024(en)
18.2 <node type specification>

- ii) Otherwise, the key label set of *ANTS* is not “omitted” and the effective key label set of *ANTS* is the key label set of *ANTS*.
- b) Otherwise, the effective key label set of *ANTS* is the key label set of *ANTS*.
- 10) If the effective key label set of *NTS* is not “omitted” and the cardinality of the effective key label set of *NTS* is less than the implementation-defined (IL003) node type key label set minimum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of node type key labels below supported minimum (42012)*.
- 11) If the effective key label set of *NTS* is not “omitted” and the cardinality of the effective key label set of *NTS* is greater than the implementation-defined (IL003) node type key label set maximum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of node type key labels exceeds supported maximum (42013)*.
- 12) If the cardinality of the label set of *NTS* is less than the implementation-defined (IL001) node label set minimum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of node labels below supported minimum (42009)*.
- 13) If the cardinality of the label set of *NTS* is greater than the implementation-defined (IL003) node label set maximum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of node labels exceeds supported maximum (42010)*.
- 14) The *property type set* of a <node type specification> *ANTS* is defined as follows.
Case:
 - a) If *ANTS* contains a <node type property types> *NTPTS*, then the property type set of *ANTS* is the property type set that is specified by the <property types specification> that constitutes *NTPTS*.
 - b) Otherwise, no <property types specification> is suitably specified and the property type set of *ANTS* is an empty property types set.
- 15) If the cardinality of the property type set of *NTS* is greater than the implementation-defined (IL002) node property set maximum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of node properties exceeds supported maximum (42011)*.
- 16) *NTS* specifies the node type with
 - a) The node type label set that is the label set of *NTS*.
 - b) The node type property type set that is the property type set of *NTS*.
- 17) For each node type *NT*, there is an implementation-defined (IV003) node type, *NTNF(NT)*, known as the normal form of *NT* (which can be *NT* itself), such that:
 - a) If *NT1* and *NT2* are two node types that have the same label sets and the same property names *PNS*, and for each property name *PN* from *PNS* it holds that the value types of the respective property types of *NT1* and *NT2* whose name is *PN* have the same normal form, then *NTNF(NT1) = NTNF(NT2)*.
 - b) *NTNF(NTNF(NT)) = NTNF(NT)*.
 - c) The results of all invocations of *NTNF(NT)* are the same.

NOTE 267 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a node type from amongst the equivalent node types are deterministic.

General Rules

- 1) A node type descriptor is created for the node type *NT* specified by *NTS* that describes *NT* and comprises:
 - a) The declared name of the primary base type of all node types (NODE DATA).

NOTE 268 — See Subclause 4.13.1, “Introduction to GQL-object types and related base types”.
 - b) The preferred name of *NT*
 - c) The node type label set of *NT*.
 - d) The node type property type set of *NT*.

Conformance Rules

- 1) Without Feature GG20, “Explicit element type names”, in conforming GQL language, a <node type specification> shall not simply contain a <node type name>.
- 2) Without Feature GG21, “Explicit element type key label sets”, in conforming GQL language, a <node type specification> shall not simply contain a <node type key label set>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

18.3 <edge type specification>

Function

Specify an edge type.

Format

```
<edge type specification> ::=
  <edge type pattern>
  | <edge type phrase>

<edge type pattern> ::=
  [ [ <edge kind> ] <edge synonym> [ TYPE ] <edge type name> ]
  { <edge type pattern directed> | <edge type pattern undirected> }

<edge type phrase> ::=
  <edge kind> <edge synonym> [ TYPE ] <edge type phrase filler> <endpoint pair phrase>

<edge type phrase filler> ::=
  <edge type name> [ <edge type filler> ]
  | <edge type filler>

<edge type filler> ::=
  <edge type key label set> [ <edge type implied content> ]
  | <edge type implied content>

<edge type implied content> ::=
  <edge type label set>
  | <edge type property types>
  | <edge type label set> <edge type property types>

<edge type key label set> ::=
  [ <label set phrase> ] <implies>

<edge type label set> ::=
  <label set phrase>

<edge type property types> ::=
  <property types specification>

<edge type pattern directed> ::=
  <edge type pattern pointing right>
  | <edge type pattern pointing left>

<edge type pattern pointing right> ::=
  <source node type reference> <arc type pointing right> <destination node type reference>

<edge type pattern pointing left> ::=
  <destination node type reference> <arc type pointing left> <source node type reference>

<edge type pattern undirected> ::=
  <source node type reference> <arc type undirected> <destination node type reference>

<arc type pointing right> ::=
  <minus left bracket> <edge type filler> <bracket right arrow>

<arc type pointing left> ::=
  <left arrow bracket> <edge type filler> <right bracket minus>

<arc type undirected> ::=
  <tilde left bracket> <edge type filler> <right bracket tilde>
```

ISO/IEC 39075:2024(en)
18.3 <edge type specification>

```
<source node type reference> ::=
    <left paren> <source node type alias> <right paren>
  | <left paren> [ <node type filler> ] <right paren>

<destination node type reference> ::=
    <left paren> <destination node type alias> <right paren>
  | <left paren> [ <node type filler> ] <right paren>

<edge kind> ::=
    DIRECTED
  | UNDIRECTED

<endpoint pair phrase> ::=
    CONNECTING <endpoint pair>

<endpoint pair> ::=
    <endpoint pair directed>
  | <endpoint pair undirected>

<endpoint pair directed> ::=
    <endpoint pair pointing right>
  | <endpoint pair pointing left>

<endpoint pair pointing right> ::=
    <left paren> <source node type alias> <connector pointing right>
    <destination node type alias> <right paren>

<endpoint pair pointing left> ::=
    <left paren> <destination node type alias> <left arrow>
    <source node type alias> <right paren>

<endpoint pair undirected> ::=
    <left paren> <source node type alias> <connector undirected>
    <destination node type alias> <right paren>

<connector pointing right> ::=
    TO
  | <right arrow>

<connector undirected> ::=
    TO
  | <tilde>

<source node type alias> ::=
    <regular identifier>

<destination node type alias> ::=
    <regular identifier>
```

Syntax Rules

- 1) Let *ETS* be the <edge type specification>.
- 2) If *ETS* is simply contained in a <closed edge reference value type>, then:
 - a) *ETS* shall not simply contain an <edge type key label set> or an <edge type name>.
 - b) *ETS* shall not simply contain a <source node type alias> or a <destination node type alias>.
 - c) *ETS* shall not simply contain a <source node type reference> that simply contains a <node type key label set>.

ISO/IEC 39075:2024(en)
18.3 <edge type specification>

- d) *ETS* shall not simply contain a <destination node type reference> that simply contains a <node type key label set>.
- 3) If *ETS* simply contains an <edge type name>, then *ETS* shall not simply contain an <edge type key label set>.
- 4) Case:
- a) If DIRECTED is simply contained in *ETS*, then *ETS* shall simply contain an <endpoint pair directed> or an <edge type pattern directed>.
- b) Otherwise, UNDIRECTED is simply contained in *ETS* and *ETS* shall simply contain an <endpoint pair undirected> or an <edge type pattern undirected>.
- 5) If *ETS* is an <edge type phrase>, then:
- a) Let *SNTA* be the <source node type alias> simply contained in *ETS*.
- b) Let *DNTA* be the <destination node type alias> simply contained in *ETS*.
- c) Let *ETF* be defined as follows
- Case:
- i) If *ETS* simply contains an implicit or explicit <edge type filler>, then *ETF* is that <edge type filler>.
- ii) Otherwise, no <edge type filler> is suitably specified and *ETF* is the zero-length character string.
- d) Case:
- i) If DIRECTED is simply contained in *ETS*, then *ETS* is equivalent to the <edge type pattern directed>:
- $$(\text{ } SNTA \text{ }) - [\text{ } ETF \text{ }] -> (\text{ } DNTA \text{ })$$
- ii) Otherwise, UNDIRECTED is simply contained in *ETS* and *ETS* is equivalent to the <edge type pattern undirected>:
- $$(\text{ } SNTA \text{ }) \sim [\text{ } ETF \text{ }] \sim (\text{ } DNTA \text{ })$$
- 6) The *key label set* of an <edge type specification> *AETS* is determined as follows.
- Case:
- a) If *AETS* simply contains an <edge type key label set> *ETKLS* that simply contains a <label set phrase> *LSP*, then the *key label set* of *AETS* is the label set specified by the *LSP*.
- b) If *AETS* simply contains an <edge type key label set> *ETKLS* that does not simply contains a <label set phrase>, then the *key label set* of *AETS* is an empty label set.
- c) If *AETS* simply contains an <edge type name> *ETN*, then the *key label set* of *AETS* is the label set specified by the <label set phrase>:
- $$:ETN$$
- d) Otherwise, no <label set phrase> is suitably specified and the *key label set* of *AETS* is “omitted”.
- 7) The *implied label set* of an <edge type specification> *AETS* is defined as follows.
- Case:

ISO/IEC 39075:2024(en)
18.3 <edge type specification>

- a) If *AETS* simply contains an <edge type label set> *ETLS*, then the implied label set of *AETS* is the label set specified by the <label set phrase> that constitutes *ETLS*.
 - b) Otherwise, no <label set phrase> is suitably specified and the implied label set of *AETS* is an empty label set.
- 8) If the key label set of *ETS* is not “omitted”, then the key label set of *ETS* and the implied label set of *ETS* shall be disjoint.
- 9) The *label set* of an <edge type specification> *AETS* is the union of the key label set of *AETS* and the implied label set of *AETS*.
- 10) The *effective key label set* of an <edge type specification> *AETS* is defined as follows.
- Case:
- a) If *AETS* is simply contained in a <graph type specification body> *GTSB*, then

Case:

 - i) If Feature GG22, “Element type key label set inference” is supported and the key label set of *AETS* is “omitted”, then:
 - 1) Let *OETSS* be the set of all <edge type specification>s simply contained in *GTSB* other than *AETS*.
 - 2) Let *OLS* be the union of the label sets of all <edge type specification>s in *OETSS*.
 - 3) Let *ULS* be the set of all labels that are in the implied label set of *AETS* but not in *OLS*.
 - 4) If *ULS* is an empty label set, then the effective key label set of *ULS* is “omitted”; otherwise, the effective key label set of *AETS* is *ULS*.
 - ii) Otherwise, the key label set of *AETS* is not “omitted” and the effective key label set of *AETS* is the key label set of *AETS*. - b) Otherwise, the effective key label set of *AETS* is the key label set of *AETS*.
- 11) If the effective key label set of *ETS* is not “omitted” and the cardinality of the effective key label set of *ETS* is less than the implementation-defined (IL003) edge type key label set minimum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of edge type key labels below supported minimum (42014)*.
- 12) If the effective key label set of *ETS* is not “omitted” and the cardinality of the effective key label set of *ETS* is greater than the implementation-defined (IL003) edge type key label set maximum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of edge type key labels exceeds supported maximum (42015)*.
- 13) If the cardinality of the label set of *ETS* is less than the implementation-defined (IL001) edge label set minimum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of edge labels below supported minimum (42006)*.
- 14) If the cardinality of the label set of *ETLS* is greater than the implementation-defined (IL001) edge label set maximum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of edge labels exceeds supported maximum (42007)*.
- 15) The *property type set* of an <edge type specification> *AETS* is defined as follows.
- Case:

ISO/IEC 39075:2024(en)
18.3 <edge type specification>

- a) If *AETS* contains an <edge type property types> *ETPTS*, then the property type set of *AETS* is the property type set that is specified by the <property types specification> that constitutes *ETPTS*.
 - b) Otherwise, no <property types specification> is suitably specified and the property type set of *AETS* is an empty property type set.
- 16) If the cardinality of the property type set of *ETS* is greater than the implementation-defined (IL002) edge property set maximum cardinality, then an exception condition is raised: *syntax error or access rule violation — number of edge properties exceeds supported maximum (42008)*.
- 17) The *source node type* of an <edge type specification> *AETS* is defined as follows.
- Case:
- a) If the <source node type reference> simply contained in *AETS* simply contains a <source node type alias> *SNTA*, then:
 - i) Let *AGTSB* be the <graph type specification body> that simply contains *AETS*.
 - ii) *AGTSB* shall simply contain a <node type specification> whose local node type alias is the same as the name specified by the <regular identifier> that constitutes *SNTA*. Let *SNTS* be that <node type specification>.
 - iii) The source node type of *AETS* is the node type specified by *SNTS*.
 - b) If the <source node type reference> simply contained in *AETS* simply contains a <node type key label set> *SNTKLS*, then:
 - i) Let *AGTSB* be the <graph type specification body> that simply contains *AETS*.
 - ii) *AGTSB* shall simply contain a <node type specification> whose key label set is the same as the label set specified by the <label set phrase> simply contained in *SNTKLS*. Let *SNT* be the node type specified by that <node type specification>.
 - iii) Let *SNTF* be the <node type filler> simply contained in the <source node type reference> simply contained in *AETS*.
 - iv) Let *ISNT* be the constraining GQL-object type of the node reference value type specified by the <closed node reference value type>:
(*SNTF*)
 - v) *SNT* and *ISNT* shall have the same normal form.
 - vi) The source node type of *AETS* is *SNT*.
 - c) Otherwise:
 - i) Let *SNTF* be defined as follows: If the <source node type reference> simply contained in *AETS* simply contains a <node type filler>, then *SNTF* is that <node type filler>; otherwise, *SNTF* is the zero-length character string.
 - ii) Let *ISNT* be the constraining GQL-object type of the node reference value type specified by the <closed node reference value type>:
(*SNTF*)
 - iii) The source node type of *AETS* is *ISNT*.
- 18) The *destination node type* of an <edge type specification> *AETS* is defined as follows.

Case:

- a) If the <destination node type reference> simply contained in *AETS* simply contains a <destination node type alias> *DNTA*, then:
 - i) Let *AGTSB* be the <graph type specification body> that simply contains *AETS*.
 - ii) *AGTSB* shall simply contain a <node type specification> whose local node type alias is the same as the name specified by the <regular identifier> that constitutes *DNTA*. Let *DNTS* be that <node type specification>.
 - iii) The destination node type of *AETS* is the node type specified by *DNTS*.
- b) If the <destination node type reference> simply contained in *AETS* simply contains a <node type key label set> *DNTKLS*, then:
 - i) Let *AGTSB* be the <graph type specification body> that simply contains *AETS*.
 - ii) *AGTSB* shall simply contain a <node type specification> whose key label set is the same as the label set specified by the <label set phrase> simply contained in *DNTKLS*. Let *DNT* be the node type specified by that <node type specification>.
 - iii) Let *DNTF* be the <node type filler> simply contained in the <destination node type reference> simply contained in *AETS*.
 - iv) Let *IDNT* be the constraining GQL-object type of the node reference value type specified by the <closed node reference value type>:
(*DNTF*)
 - v) *DNT* and *IDNT* shall have the same normal form.
 - vi) The destination node type of *AETS* is *DNT*.
- c) Otherwise:
 - i) Let *DNTF* be defined as follows: If the <destination node type reference> simply contained in *AETS* simply contains a <node type filler>, then *DNTF* is that <node type filler>; otherwise, *DNTF* is the zero-length character string.
 - ii) Let *IDNT* be the constraining GQL-object type of the node reference value type specified by the <closed node reference value type>:
(*DNTF*)
 - iii) The destination node type of *AETS* is *IDNT*.

19) *ETS* specifies an edge type as follows.

Case:

- a) If *ETS* simply contains an <edge type pattern directed>, then *ETS* specifies the directed edge type with:
 - i) The edge type label set that is the label set of *ETS*.
 - ii) The edge type property type set that is the property type set of *ETS*.
 - iii) The source node type that is the source node type of *ETS*.
 - iv) The destination node type that is the destination node type of *ETS*.

- b) Otherwise, *ETS* simply contains an <edge type pattern undirected> and *ETS* specifies the undirected edge type with:
- The edge type label set that is the label set of *ETS*.
 - The edge type property type set that is the property type set of *ETS*.
 - The set of endpoint node types that is the set comprising the source node type of *ETS* and the destination node type of *ETS*.
- 20) For each edge type *ET*, there is an implementation-defined (IV003) edge type, *ETNF(ET)*, known as the *normal form* of *ET* (which can be *ET* itself), such that:
- If *ET1* and *ET2* are two directed edge types that have the same label sets, the same property names *PNS*, respective source node types with the same normal form, and respective destination node types with the same normal form, and for each property name *PN* from *PNS* it holds that the value types of the respective property types of *ET1* and *ET2* whose name is *PN* have the same normal form, then $ETNF(ET1) = ETNF(ET2)$.
 - If *ET1* and *ET2* are two undirected edge types that have the same label sets, the same property names *PNS*, and the same sets of normal forms of their respective endpoint node types, and for each property name *PN* from *PNS* it holds that the value types of the respective property types of *ET1* and *ET2* whose name is *PN* have the same normal form, then $ETNF(ET1) = ETNF(ET2)$.
 - $ETNF(ETNF(ET)) = ETNF(ET)$.
 - The results of all invocations of *ETNF(ET)* are the same.

NOTE 269 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of an edge type from amongst the equivalent edge types are deterministic.

General Rules

- 1) An edge type descriptor is created for the edge type *ET* specified by *ETS* that describes *ET* and comprises:
- The declared name of the primary base type of all edge types (EDGE DATA).
NOTE 270 — See Subclause 4.13.1, “Introduction to GQL-object types and related base types”.
 - The preferred name of *ET*.
 - The edge type label set of *ET*.
 - The edge type property type set of *ET*.
 - Case:
 - If *ET* is directed, then:
 - The indication that the edge type is directed.
 - The source node type of *ET*.
 - The destination node type of *ET*.
 - Otherwise:
 - The indication that the edge type is undirected.
 - The set of endpoint node types of *ET*.

Conformance Rules

- 1) Without Feature GH02, “Undirected edge patterns”, conforming GQL language shall not contain an <edge type specification> that simply contains an <edge kind> that is UNDIRECTED, an <endpoint pair> that is an <endpoint pair undirected>, or an <edge type pattern undirected>.
- 2) Without Feature GG20, “Explicit element type names”, in conforming GQL language, an <edge type specification> shall not simply contain an <edge type name>.
- 3) Without Feature GG21, “Explicit element type key label sets”, in conforming GQL language, an <edge type specification> shall not simply contain an <edge type key label set> or a <node type key label set>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

18.4 <label set phrase> and <label set specification>

Function

Specify label sets.

Format

```
<label set phrase> ::=  
    LABEL <label name>  
    | LABELS <label set specification>  
    | <is or colon> <label set specification>  
  
<label set specification> ::=  
    <label name> [ { <ampersand> <label name> }... ]
```

Syntax Rules

- 1) If a <label set phrase> *LSP* simply contains LABEL and the <label name> *LN*, then *LSP* is equivalent to:

 :*LN*
- 2) Every <label set phrase> *LSP* specifies the label set specified by the implicit or explicit <label set specification> immediately contained in *LSP*.
- 3) Every <label set specification> *LSS* specifies the set of distinct label names specified by the <label name>s simply contained in *LSS*.

General Rules

None.

Conformance Rules

None.

18.5 <property types specification>

Function

Specify property types.

Format

```
<property types specification> ::=  
  <left brace> [ <property type list> ] <right brace>  
  
<property type list> ::=  
  <property type> [ { <comma> <property type> }... ]
```

Syntax Rules

- 1) For every <property type> *PT* simply contained in a <property types specification> *PTSS*, the name of *PT* shall be different from the name of every other <property type> simply contained in *PTSS*.
- 2) Every <property types specification> specifies the property types specified by the <property type list> that it immediately contains.
- 3) Every <property type list> *PTL* specifies the property types that are specified by the <property type>s immediately contained in *PTL*.

General Rules

None.

Conformance Rules

None.

18.6 <property type>

Function

Define a property type.

Format

```
<property type> ::=  
  <property name> [ <typed> ] <property value type>
```

Syntax Rules

- 1) The *name* of a <property type> *APT* is the name specified by the <property name> simply contained in *APT*.
- 2) The *value type* of a <property type> *APT* is the value type specified by the <property value type> simply contained in *APT*.
- 3) The property type specified by the <property type> *TPT* is the pair comprising the name of *TPT* and the value type of *TPT*.
- 4) The value type of the property type specified by a <property type> shall be a supported property value type.

NOTE 271 — See Subclause 4.3.4, “Properties and supported property value types”.

General Rules

- 1) A data type descriptor is created that describes the value type of the property type being defined.
- 2) A property type descriptor is created that describes the property type being defined. The property type descriptor includes the following:
 - a) The name of the property type.
 - b) The value type of the property type.

Conformance Rules

- 1) Without Feature GV48, “Nested record types”, conforming GQL language shall not contain a <property type> that simply contains a <record type>.

18.7 <property value type>

Function

Specify a property value type.

Format

<property value type> ::=
 <value type>

Syntax Rules

- 1) Let *PV* be the <property value type>.
- 2) Let *VT* be the <value type> immediately contained in *PV*.
- 3) *PV* specifies the property value type that is specified by *VT*.
- 4) The property value type specified by *PV* shall be a supported property value type.

NOTE 272 — See Subclause 4.3.4, “Properties and supported property value types”, for the definition of supported property value type.

General Rules

None.

Conformance Rules

None.

18.8 <binding table type>

Function

Define a binding table type.

Format

```
<binding table type> ::=  
[ BINDING ] TABLE <field types specification>
```

Syntax Rules

- 1) Every <binding table type> *BTT* specifies a material binding table type whose record type is the closed material record type whose field types are specified by the <field types specification> immediately contained in *BTT*.
- 2) For each binding table type *BTT*, there is an implementation-defined (IV003) binding table type, *BTNF(BTT)*, known as the *normal form* of *BTT* (which can be *BTT* itself), such that:
 - a) If *BTT1* and *BTT2* are two binding table types whose record types have the same normal form and whose indications regarding the inclusion of the null value are the same, then *BTNF(BTT1) = BTNF(BTT2)*.
 - b) *BTNF(BTNF(BTT)) = BTNF(BTT)*.
 - c) The results of all invocations of *BTNF(BTT)* are the same.

NOTE 273 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a binding table type from amongst the equivalent binding table types are deterministic.

General Rules

- 1) Let *BTT* be the binding table type specified by <binding table type>.
- 2) A record data type descriptor for the record type *RT* of *BTT* is created that comprises:
 - a) The base type name of all record types (RECORD DATA).
 - b) The indication that *RT* is closed.
 - c) A field type descriptor for every <field type> simply contained in <binding table type>, according to the Syntax Rules and General Rules of Subclause 18.10, "<field type>", applied to the <field type>s in the order in which they were specified.
 - d) The indication that *RT* excludes the null value.
- 3) A binding table data type descriptor is created that comprises:
 - a) The base type name of all binding table types (BINDING TABLE DATA).
 - b) The record data type descriptor *RT*.

Conformance Rules

None.

18.9 <value type>

Function

Specify a value type.

Format

```
<value type> ::=
    <predefined type>
  | <constructed value type>
  | <dynamic union type>

<typed> ::=
    <double colon> | TYPED

<predefined type> ::=
    <boolean type>
  | <character string type>
  | <byte string type>
  | <numeric type>
  | <temporal type>
  | <reference value type>
  | <immaterial value type>

<boolean type> ::=
    { BOOL | BOOLEAN } [ <not null> ]

<character string type> ::=
    STRING [ <left paren> [ <min length> <comma> ] <max length> <right paren> ]
    [ <not null> ]
  | CHAR [ <left paren> <fixed length> <right paren> ] [ <not null> ]
  | VARCHAR [ <left paren> <max length> <right paren> ] [ <not null> ]

<byte string type> ::=
    BYTES [ <left paren> [ <min length> <comma> ] <max length> <right paren> ]
    [ <not null> ]
  | BINARY [ <left paren> <fixed length> <right paren> ] [ <not null> ]
  | VARBINARY [ <left paren> <max length> <right paren> ] [ <not null> ]

<min length> ::=
    <unsigned integer>

<max length> ::=
    <unsigned integer>

<fixed length> ::=
    <unsigned integer>

<numeric type> ::=
    <exact numeric type>
  | <approximate numeric type>

<exact numeric type> ::=
    <binary exact numeric type>
  | <decimal exact numeric type>

<binary exact numeric type> ::=
    <signed binary exact numeric type>
  | <unsigned binary exact numeric type>
```

ISO/IEC 39075:2024(en)
18.9 <value type>

```
<signed binary exact numeric type> ::=
  INT8 [ <not null> ]
| INT16 [ <not null> ]
| INT32 [ <not null> ]
| INT64 [ <not null> ]
| INT128 [ <not null> ]
| INT256 [ <not null> ]
| SMALLINT [ <not null> ]
| INT [ <left paren> <precision> <right paren> ] [ <not null> ]
| BIGINT [ <not null> ]
| [ SIGNED ] <verbose binary exact numeric type>

<unsigned binary exact numeric type> ::=
  UINT8 [ <not null> ]
| UINT16 [ <not null> ]
| UINT32 [ <not null> ]
| UINT64 [ <not null> ]
| UINT128 [ <not null> ]
| UINT256 [ <not null> ]
| USMALLINT [ <not null> ]
| UINT [ <left paren> <precision> <right paren> ] [ <not null> ]
| UBIGINT [ <not null> ]
| UNSIGNED <verbose binary exact numeric type>

<verbose binary exact numeric type> ::=
  INTEGER8 [ <not null> ]
| INTEGER16 [ <not null> ]
| INTEGER32 [ <not null> ]
| INTEGER64 [ <not null> ]
| INTEGER128 [ <not null> ]
| INTEGER256 [ <not null> ]
| SMALL INTEGER [ <not null> ]
| INTEGER [ <left paren> <precision> <right paren> ] [ <not null> ]
| BIG INTEGER [ <not null> ]

<decimal exact numeric type> ::=
  { DECIMAL | DEC } [ <left paren> <precision> [ <comma> <scale> ] <right paren>
    [ <not null> ] ]

<precision> ::=
  <unsigned decimal integer>

<scale> ::=
  <unsigned decimal integer>

<approximate numeric type> ::=
  FLOAT16 [ <not null> ]
| FLOAT32 [ <not null> ]
| FLOAT64 [ <not null> ]
| FLOAT128 [ <not null> ]
| FLOAT256 [ <not null> ]
| FLOAT [ <left paren> <precision> [ <comma> <scale> ] <right paren> ] [ <not null> ]
| REAL [ <not null> ]
| DOUBLE [ PRECISION ] [ <not null> ]

<temporal type> ::=
  <temporal instant type>
| <temporal duration type>

<temporal instant type> ::=
  <datetime type>
| <localdatetime type>
| <date type>
```

```
| <time type>
| <localtime type>

<datetime type> ::=
    ZONED DATETIME [ <not null> ]
    | TIMESTAMP WITH TIME ZONE [ <not null> ]

<localdatetime type> ::=
    LOCAL DATETIME [ <not null> ]
    | TIMESTAMP [ WITHOUT TIME ZONE ] [ <not null> ]

<date type> ::=
    DATE [ <not null> ]

<time type> ::=
    ZONED TIME [ <not null> ]
    | TIME WITH TIME ZONE [ <not null> ]

<localtime type> ::=
    LOCAL TIME [ <not null> ]
    | TIME WITHOUT TIME ZONE [ <not null> ]

<temporal duration type> ::=
    DURATION <left paren> <temporal duration qualifier> <right paren> [ <not null> ]

<temporal duration qualifier> ::=
    YEAR TO MONTH
    | DAY TO SECOND

<reference value type> ::=
    <graph reference value type>
    | <binding table reference value type>
    | <node reference value type>
    | <edge reference value type>

<immaterial value type> ::=
    <null type>
    | <empty type>

<null type> ::=
    NULL

<empty type> ::=
    NULL <not null>
    | NOTHING

<graph reference value type> ::=
    <open graph reference value type>
    | <closed graph reference value type>

<closed graph reference value type> ::=
    [ PROPERTY ] GRAPH <nested graph type specification> [ <not null> ]

<open graph reference value type> ::=
    [ ANY ] [ PROPERTY ] GRAPH [ <not null> ]

<binding table reference value type> ::=
    <binding table type> [ <not null> ]

<node reference value type> ::=
    <open node reference value type>
    | <closed node reference value type>

<closed node reference value type> ::=
    <node type specification> [ <not null> ]
```

<open node reference value type> ::=
 [ANY] <node synonym> [<not null>]

<edge reference value type> ::=
 <open edge reference value type>
 | <closed edge reference value type>

<closed edge reference value type> ::=
 <edge type specification> [<not null>]

<open edge reference value type> ::=
 [ANY] <edge synonym> [<not null>]

<constructed value type> ::=
 <path value type>
 | <list value type>
 | <record type>

<path value type> ::=
 PATH [<not null>]

<list value type> ::=
 {
 <list value type name> <left angle bracket> <value type> <right angle bracket>
 | [<value type>] <list value type name>
 } [<left bracket> <max length> <right bracket>] [<not null>]

<list value type name> ::=
 [GROUP] <list value type name synonym>

<list value type name synonym> ::=
 LIST | ARRAY

<record type> ::=
 [ANY] RECORD [<not null>]
 | [RECORD] <field types specification> [<not null>]

<field types specification> ::=
 <left brace> [<field type list>] <right brace>

<field type list> ::=
 <field type> [{ <comma> <field type> }...]

<dynamic union type> ::=
 <open dynamic union type>
 | <dynamic property value type>
 | <closed dynamic union type>

<open dynamic union type> ::=
 ANY [VALUE] [<not null>]

<dynamic property value type> ::=
 [ANY] PROPERTY VALUE [<not null>]

<closed dynamic union type> ::=
 ANY [VALUE] <left angle bracket> <component type list> <right angle bracket>
 | <component type list>

<component type list> ::=
 <component type> [{ <vertical bar> <component type> }...]

<component type> ::=
 <value type>

<not null> ::=

NOT NULL

Syntax Rules

- 1) Every <value type> specifies the value type that is the data type specified by the immediately contained BNF non-terminal instance.
- 2) Every <predefined type> specifies the data type specified by the immediately contained BNF non-terminal instance.
- 3) Every <reference value type> specifies the reference value type that is the data type specified by the immediately contained <graph reference value type>, <binding table reference value type>, <node reference value type>, or <edge reference value type>.
- 4) The nullability specified by a BNF non-terminal instance *NT* that is a <value type> but not a <closed dynamic union type> is defined as follows.

Case:

- a) If *NT* contains <not null> without an intervening instance of <value type>, then *NT* specifies known not nullable.
- b) If *NT* is NOTHING, then *NT* specifies known not nullable.
- c) Otherwise, *NT* specifies possibly nullable.

NOTE 274 — See Subclause 4.17.4, “Nullability”.

- 5) Every <boolean type> *BT* specifies the *Boolean type* with the nullability specified by *BT*.
- 6) For each Boolean type *BT*, there is an implementation-defined (IV008) Boolean type *BNF(BT)*, known as the *normal form* of *BT* (which can be *BT* itself), such that:
 - a) The nullability of *BNF(BT)* is the nullability of *BT*.
 - b) The declared name of *BNF(BT)* is the preferred name of Boolean types.
 - c) All invocations of *BNF(BT)* are the same.

NOTE 275 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a Boolean type from amongst the equivalent Boolean types are deterministic.

- 7) The value of every <max length> shall be greater than or equal to 1 (one).
- 8) The value of every <min length> shall be greater than or equal to 0 (zero).
- 9) If <min length> is omitted, then a <min length> of 0 (zero) is implicit.
- 10) The value of every <fixed length> shall be greater than or equal to 1 (one).
- 11) If <fixed length> is omitted, then a <fixed length> of 1 (one) is implicit.
- 12) Every <character string type> *CST* specifies the *character string type* with the nullability specified by *CST*.
- 13) If both <min length> *MINL* and <max length> *MAXL* are specified or implicit in a <character string type>, then *MINL* shall be less than or equal to *MAXL*.
- 14) If <min length> *CSMINL* is specified or implicit in a <character string type> *CST*, then the minimum length of the character string type specified by *CST* is the value of *CSMINL*.
- 15) If <max length> *CSMAL* is specified in a <character string type> *CST*, then the maximum length of the character string type specified by *CST* is the value of *CSMAXL*.

- 16) If <fixed length> *CSFIXL* is specified or implicit in a <character string type> *CST*, then the minimum length of the character string type specified by *CST* is the value of *CSFIXL* and the maximum length of the character string type specified by *CST* is the value of *CSFIXL*.
- 17) If neither <max length> nor <fixed length> are specified or implicit in a <character string type> *CST*, then the maximum length of the character string type specified by *CST* is implementation-defined (IL013) but shall be greater than or equal to $2^{14}-1 = 16383$ characters.
- 18) For each character string type *CST*, there is an implementation-defined (IV008) character string type *CSNF(CST)*, known as the *normal form* of *CST* (which can be *CST* itself), such that *CSNF(CST)* and *CST* have the same minimum length and the same maximum length and the nullability of *CSNF(CST)* is the nullability of *CST* and

Case:

- a) If *CST* is a fixed-length character string type, then:

- i) The declared name of *CSNF(CST)* is the preferred name of fixed-length character string types.
- ii) All invocations of *CSNF(CST)* are the same.

NOTE 276 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a fixed-length character string type from amongst the equivalent fixed-length character string types are deterministic.

- b) If *CST* is a variable-length character string type, then:

- i) The declared name of *CSNF(CST)* is the preferred name of variable-length character string types.
- ii) All invocations of *CSNF(CST)* are the same.

NOTE 277 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a variable-length character string type from amongst the equivalent variable-length character string types are deterministic.

- 19) The material values of a character string type with minimum length *CSMINL* and maximum length *CSMAXL* are the character strings whose length is both greater than or equal to *CSMINL* and less than or equal to *CSMAXL*.
- 20) Every <byte string type> *BST* specifies the *byte string type* with the nullability specified by *BST*.
- 21) If both <min length> *MINL* and <max length> *MAXL* are specified or implicit in a <byte string type>, then *MINL* shall be less than or equal to *MAXL*.
- 22) If <min length> *BSMINL* is specified or implicit in a <byte string type> *BST*, then the minimum length of the byte string type specified by *BST* is the value of *BSMINL*.
- 23) If <max length> *BSMAXL* is specified in a <byte string type> *BST*, then the maximum length of the byte string type specified by *BST* is the value of *BSMAXL*.
- 24) If <fixed length> *BSFIXL* is specified or implicit in a <byte string type> *BST*, then the minimum length of the byte string type specified by *BST* is the value of *BSFIXL* and the maximum length of the byte string type specified by *BST* is the value of *BSFIXL*.
- 25) If neither <max length> nor <fixed length> are specified or implicit in a <byte string type> *BST*, then the maximum length of the byte string type specified by *BST* is implementation-defined (IL013) but shall be greater than or equal to $2^{16}-2 = 65534$ bytes.
- 26) For each byte string type *BST*, there is an implementation-defined (IV008) byte string type *BSNF(BST)*, known as the *normal form* of *BST* (which can be *BST* itself), such that *BSNF(BST)* and

BST have the same minimum length and the same maximum length and the nullability of *BSNF*(*BST*) is the nullability of *BST*.

Case:

a) If *BST* is a fixed-length byte string type, then:

- i) The declared name of *BSNF*(*BST*) is the preferred name of fixed-length byte string types.
- ii) All invocations of *BSNF*(*BST*) are the same.

NOTE 278 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a fixed-length byte string type from amongst the equivalent fixed-length byte string types are deterministic.

b) If *BST* is a variable-length byte string type, then:

- i) The declared name of *BSNF*(*BST*) is the preferred name of variable-length byte string types.
- ii) All invocations of *BSNF*(*BST*) are the same.

NOTE 279 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of a variable-length byte string type from amongst the equivalent variable-length byte string types are deterministic.

- 27) The material values of a byte string type with minimum length *BSMINL* and maximum length *BSMAXL* are the byte strings whose length is both greater than or equal to *BSMINL* and less than or equal to *BSMAXL*.
- 28) The minimum length of every byte string type *BST* shall be less than or equal to the maximum length of *BST*.
- 29) The value of every <precision> shall be greater than or equal to 1 (one).
- 30) Every <numeric type> specifies the *numeric type* specified by the immediately contained <exact numeric type> or the immediately contained <approximate numeric type>.
- 31) For each exact numeric type *ENT*, there is an implementation-defined (IV008) exact numeric type, *ENNF*(*ENT*), known as the *normal form* of *ENT* (which can be *ENT* itself), such that:
 - a) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either SIGNED INTEGER, INTEGER, or INT, then *ENNF*(*ENT1*) is the same as *ENNF*(*ENT2*).
 - b) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either SIGNED SMALL INTEGER, SMALL INTEGER, or SMALLINT, then *ENNF*(*ENT1*) is the same as *ENNF*(*ENT2*).
 - c) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either SIGNED BIG INTEGER, BIG INTEGER, or BIGINT, then *ENNF*(*ENT1*) is the same as *ENNF*(*ENT2*).
 - d) For each *p* from the set {16, 32, 64, 128, 256}, if *ENT1* and *ENT2* are exact numeric types whose declared names individually are either SIGNED INTEGER_{*p*}, INTEGER_{*p*}, or INT_{*p*}, then *ENNF*(*ENT1*) is the same as *ENNF*(*ENT2*).
 - e) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either UNSIGNED INTEGER or UINT, then *ENNF*(*ENT1*) is the same as *ENNF*(*ENT2*).
 - f) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either UNSIGNED SMALL INTEGER, or USMALLINT, then *ENNF*(*ENT1*) is the same as *ENNF*(*ENT2*).

- g) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either UNSIGNED BIG INTEGER, or UBIGINT, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
- h) For each *p* from the set {16, 32, 64, 128, 256}, if *ENT1* and *ENT2* are exact numeric types whose declared names individually are either UNSIGNED INTEGER_{*p*} or UINT_{*p*}, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
- i) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either DEC or DECIMAL, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
- j) The precision, scale, and radix of *ENNF(ENT)* are the same as the precision, scale, and radix, respectively, of *ENT*.

NOTE 280 — The precision, scale, and radix are determined when an exact numeric type is specified prior to the construction of its descriptor.

- k) The nullability of *ENNF(ENT)* is the nullability of *ENT*.
- l) *ENNF(ENNF(ENT)) = ENNF(ENT)*.
- m) The results of all invocations of *ENNF(ENT)* are the same.

NOTE 281 — This ensures that all invocations of the implementation-defined (IV008) choice of the normal form of an exact numeric type from amongst the equivalent exact numeric types are deterministic.

32) For the <exact numeric type>s:

- a) The maximum value of a <precision> is implementation-defined (IL011). <precision> shall not be greater than this value.
- b) The maximum value of a <scale> is implementation-defined (IL011). <scale> shall not be greater than this maximum value.

33) Every <exact numeric type> specifies the *exact numeric type* specified by the immediately contained <binary exact numeric type> or the immediately contained <decimal exact numeric type>.

34) Every <binary exact numeric type> specifies the *binary exact numeric type* specified by the immediately contained <signed binary exact numeric type> or the immediately contained <unsigned binary exact numeric type>.

35) Every <signed binary exact numeric type> *BESNT* specifies a *signed binary exact numeric type*.

Case:

- a) If *BESNT* starts with SIGNED INTEGER8, INTEGER8, or INT8, then *BESNT* specifies the *signed 8-bit integer type* with precision of 7 and with scale 0 (zero) and with the nullability specified by *BESNT*.
- b) If *BESNT* starts with SIGNED INTEGER16, INTEGER16, or INT16, then *BESNT* specifies the *signed 16-bit integer type* with precision of 15 and with scale 0 (zero) and with the nullability specified by *BESNT*.
- c) If *BESNT* starts with SIGNED INTEGER32, INTEGER32, or INT32, then *BESNT* specifies the *signed 32-bit integer type* with precision of 31 and with scale 0 (zero) and with the nullability specified by *BESNT*.
- d) If *BESNT* starts with SIGNED INTEGER64, INTEGER64, or INT64, then *BESNT* specifies the *signed 64-bit integer type* with precision of 63 and with scale 0 (zero) and with the nullability specified by *BESNT*.

- e) If *BESNT* starts with SIGNED INTEGER128, INTEGER128, or INT128, then *BESNT* specifies the *signed 128-bit integer type* with precision of 127 and with scale 0 (zero) and with the nullability specified by *BESNT*.
 - f) If *BESNT* starts with SIGNED INTEGER256, INTEGER256, or INT256, then *BESNT* specifies the *signed 256-bit integer type* with precision of 255 and with scale 0 (zero) and with the nullability specified by *BESNT*.
 - g) If *BESNT* starts with SIGNED INTEGER, INTEGER, or INT, then *BESNT* specifies the *signed regular integer type* with implementation-defined (ID028) precision greater than or equal to 31 and with scale 0 (zero) and with the nullability specified by *BESNT*.
 - h) If *BESNT* starts with SIGNED SMALL INTEGER, SMALL INTEGER, or SMALLINT, then *BESNT* specifies the *signed small integer type* with implementation-defined (ID028) precision less than or equal to the precision of the signed regular integer type and with scale 0 (zero) and with the nullability specified by *BESNT*.
 - i) If *BESNT* starts with SIGNED BIG INTEGER, BIG INTEGER, or BIGINT, then *BESNT* specifies the *signed big integer type* with implementation-defined (ID028) precision greater than or equal to the precision of the signed regular integer type and with scale 0 (zero) and with the nullability specified by *BESNT*.
 - j) Otherwise, *BESNT* specifies the *signed user-specified integer type* with implementation-defined (ID028) precision greater than or equal to the value of the <precision> immediately contained in *BESNT* as its binary precision in bits and with scale 0 (zero) and with the nullability specified by *BESNT*.
- 36) Every <unsigned binary exact numeric type> *BEUNT* specifies an *unsigned binary exact numeric type*.
- Case:
- a) If *BEUNT* starts with UNSIGNED INTEGER8 or UINT8, then *BEUNT* specifies the *unsigned 8-bit integer type* with precision of 8 and with scale 0 (zero) and with the nullability specified by *BEUNT*.
 - b) If *BEUNT* starts with UNSIGNED INTEGER16 or UINT16, then *BEUNT* specifies the *unsigned 16-bit integer type* with precision of 16 and with scale 0 (zero) and with the nullability specified by *BEUNT*.
 - c) If *BEUNT* starts with UNSIGNED INTEGER32 or UINT32, then *BEUNT* specifies the *unsigned 32-bit integer type* with precision of 32 and with scale 0 (zero) and with the nullability specified by *BEUNT*.
 - d) If *BEUNT* starts with UNSIGNED INTEGER64 or UINT64, then *BEUNT* specifies the *unsigned 64-bit integer type* with precision of 64 and with scale 0 (zero) and with the nullability specified by *BEUNT*.
 - e) If *BEUNT* starts with UNSIGNED INTEGER128 or UINT128, then *BEUNT* specifies the *unsigned 128-bit integer type* with precision of 128 and with scale 0 (zero) and with the nullability specified by *BEUNT*.
 - f) If *BEUNT* starts with UNSIGNED INTEGER256 or UINT256, then *BEUNT* specifies the *unsigned 256-bit integer type* with precision of 256 and with scale 0 (zero) and with the nullability specified by *BEUNT*.
 - g) If *BEUNT* starts with UNSIGNED INTEGER or UINT, then *BEUNT* specifies the *unsigned regular integer type* with the same precision as the precision of the unsigned 32-bit integer type and with scale 0 (zero) and with the nullability specified by *BEUNT*.

- h) If *BEUNT* starts with UNSIGNED SMALL INTEGER or USMALLINT, then *BEUNT* specifies the *unsigned small integer type* with implementation-defined (ID028) precision less than or equal to the precision of the unsigned regular integer type and with scale zero (0) and with the nullability specified by *BEUNT*.
 - i) If *BEUNT* starts with UNSIGNED BIG INTEGER or UBIGINT, then *BEUNT* specifies the *unsigned big integer type* with implementation-defined (ID028) precision greater than or equal to the precision of the unsigned regular integer type and with scale zero (0) and with the nullability specified by *BEUNT*.
 - j) Otherwise, *BEUNT* specifies the *unsigned user-specified integer type* with implementation-defined (ID028) precision greater than or equal to the value of the <precision> immediately contained in *BEUNT* as its binary precision in bits and with scale 0 (zero) and with the nullability specified by *BEUNT*.
- 37) The value of every <scale> shall be greater than or equal to 0 (zero).
- 38) If an <exact numeric type> *ENT* contains the <scale> *SCALE*, then the value of *SCALE* shall not be greater than the value of the <precision> contained in *ENT*.
- 39) Every <decimal exact numeric type> *DENT* specifies an exact numeric type with decimal precision.
- Case:
- a) If *DENT* immediately contains DECIMAL or DEC and the <precision> *PREC* and the <scale> *SCALE*, then *DENT* specifies the *user-specified decimal exact numeric type* with implementation-defined (ID034) decimal precision in digits greater than or equal to the decimal precision in digits specified by *PREC* and with implementation-defined (ID037) decimal scale in digits specified by *SCALE* and with the nullability specified by *DENT*.
 - b) If *DENT* immediately contains DECIMAL or DEC and the <precision> *PREC* but no <scale>, then *DENT* specifies the *user-specified decimal exact numeric type* with implementation-defined (ID034) decimal precision in digits greater than or equal to the decimal precision in digits specified by *PREC* and with scale 0 (zero) and with the nullability specified by *DENT*.
 - c) Otherwise, *DENT* specifies the *regular decimal exact numeric type* with implementation-defined (ID034) decimal precision and with scale 0 (zero) and with the nullability specified by *DENT*.
- 40) The value of every <precision> contained in an <approximate numeric type> shall be equal to or greater than 2.
- NOTE 282 — This accounts for the possible inclusion of a leading bit in the precision describing the size of the mantissa of an approximate numeric value that is implied by but not included in the underlying physical representation.
- 41) For each approximate numeric type *ANT*, there is an implementation-defined (IV008) approximate numeric type *ANNF(ANT)*, known as the *normal form* of *ANT* (which can be *ANT* itself), such that:
- a) The precision and scale of *ANNF(ANT)* are the same as the precision and scale, respectively, of *ANT*.
- NOTE 283 — The precision and scale are determined when an exact numeric type is specified prior to the construction of its descriptor.
- b) The nullability of *ANNF(ANT)* is the nullability of *ANT*.
 - c) If *ANT1* and *ANT2* are exact numeric types whose declared names individually are either DOUBLE or DOUBLE PRECISION, then *ANNF(ANT1)* is the same as *ANNF(ANT2)*.
 - d) *ANNF(ANNF(ANT)) = ANNF(ANT)*.
 - e) The results of all invocations of *ANNF(ANT)* are the same.

NOTE 284 — This ensures that all invocations of the implementation-defined (IV008) choice of the normal form of an approximate numeric type from amongst the equivalent approximate numeric types are deterministic.

42) For the <approximate numeric type>s:

- a) The maximum value of a <precision> is implementation-defined (IL011). <precision> shall not be greater than this value.
- b) The maximum value of a <scale> is implementation-defined (IL011). <scale> shall not be greater than this maximum value.

43) Every <approximate numeric type> *ANT* specifies an *approximate numeric type*:

NOTE 285 — The precision and scale of an approximate numeric type specify the number of most significant binary digits except for the sign of the mantissa and, respectively, the exponent of approximate numbers of the type.

NOTE 286 — Every finite number of an approximate numeric type of precision p and scale s can be specified using an <approximate numeric literal> that contains at most $\lfloor \log_{10}(2^p) \rfloor$ decimal digits before the decimal point and at most $\lfloor \log_{10}(2^s) \rfloor$ decimal digits after the decimal point.

Case:

- a) If *ANT* starts with FLOAT16, then *ANT* specifies the 16-bit *approximate numeric type* with precision 10 and with scale 4 and with the nullability specified by *ANT*.
- b) If *ANT* starts with FLOAT32, then *ANT* specifies the 32-bit *approximate numeric type* with precision of 23 and with scale 7 and with the nullability specified by *ANT*.
- c) If *ANT* starts with FLOAT64, then *ANT* specifies the 64-bit *approximate numeric type* with precision of 52 and with scale 10 and with the nullability specified by *ANT*.
- d) If *ANT* starts with FLOAT128, then *ANT* specifies the 128-bit *approximate numeric type* with precision of 112 and with scale 14 and with the nullability specified by *ANT*.
- e) If *ANT* starts with FLOAT256, then *ANT* specifies the 256-bit *approximate numeric type* with precision of 236 and with scale 18 and with the nullability specified by *ANT*.
- f) If *ANT* starts with FLOAT, then *ANT* specifies the *regular approximate numeric type* with implementation-defined (ID037) precision greater than or equal to 23 and with implementation-defined (ID037) scale greater than or equal to 7 and with the nullability specified by *ANT*.
- g) If *ANT* starts with REAL, then *ANT* specifies the *real approximate numeric type* with an implementation-defined (ID037) precision less than or equal to the precision of the regular approximate numeric type and with implementation-defined (ID037) scale and with the nullability specified by *ANT*.
- h) If *ANT* starts with DOUBLE or DOUBLE PRECISION, then *ANT* specifies the *double approximate numeric type* with implementation-defined (ID037) precision greater than or equal to the precision of the regular approximate numeric type and with implementation-defined (ID037) scale and with the nullability specified by *ANT*.
- i) Otherwise, *ANT* specifies a user-specified approximate numeric type:

Case:

- i) If *ANT* immediately contains the <precision> *PREC* but no scale, then *ANT* specifies the *user-specified approximate numeric type* with implementation-defined (ID037) precision greater than or equal to *PREC* and with implementation-defined (ID037) scale and with the nullability specified by *ANT*.

- ii) Otherwise, *ANT* immediately contains the <precision> *PREC* and the <scale> *SCALE*, then *ANT* specifies the user-specified approximate numeric type with implementation-defined (ID037) precision greater than or equal to *PREC* and with implementation-defined (ID037) scale greater than or equal to *SCALE* and with the nullability specified by *ANT*.

- 44) The preferred name of every numeric type is the declared name of its normal form.
- 45) If a <numeric type> *NT* contains the <precision> *PREC*, then *PREC* is the explicitly specified precision of the numeric type specified by *NT*.
- 46) If a <numeric type> *NT* contains the <scale> *SCALE*, then *SCALE* is the explicitly specified scale of the numeric type specified by *NT*.
- 47) Every <temporal type> specifies the temporal type specified by the immediately contained <datetime type>, <localdatetime type>, <date type>, <time type>, <localtime type>, or <temporal duration type>.
- 48) Every <datetime type> *DTT* specifies the zoned datetime type with the nullability specified by *DTT*.
- 49) Every <localdatetime type> *LDT* specifies the local datetime type with the nullability specified by *LDT*.
- 50) Every <date type> *DT* specifies the date type with the nullability specified by *DT*.
- 51) Every <time type> *TT* specifies the zoned time type with the nullability specified by *TT*.
- 52) Every <localtime type> *LT* specifies the local time type with the nullability specified by *LT*.
- 53) For each temporal instant type *TINT*, there is an implementation-defined (IV008) temporal instant type, *TINTNF*(*TINT*), known as the *normal form* of *TINT* (which can be *TINT* itself), such that:
 - a) If *TINT1* and *TINT2* are two temporal instant types whose indications regarding whether their values capture the date, indications regarding whether their values capture the time of day, indications regarding whether their values include a time zone displacement, and indications regarding the inclusion of the null value are all the same, then *TINTNF*(*TINT1*) = *TINTNF*(*TINT2*).
 - b) *TINTNF*(*TINTNF*(*TINT*)) = *TINTNF*(*TINT*).
 - c) The results of all invocations of *TINTNF*(*TINT*) are the same.

NOTE 287 — This ensures that all invocations of the implementation-defined (IV008) choice of the normal form of a temporal instant type from amongst the equivalent exact numeric types are deterministic.

- 54) If the <temporal duration qualifier> *DQ* is specified, then
 - Case:
 - a) If *DQ* is YEAR TO MONTH, then *DQ* specifies the year and month-based duration unit group.
 - b) Otherwise, *DQ* is DAY TO SECOND and *DQ* specifies the day and time-based duration unit group.

NOTE 288 — See Subclause 4.16.6.3, “Temporal duration types” regarding temporal duration unit groups.

- 55) Every <temporal duration type> *DUT* specifies the temporal duration type with the temporal duration unit group specified by *DUT* and with the nullability specified by *DUT*.
- 56) Every <graph reference value type> specifies the graph reference value type specified by its immediately contained <open graph reference value type> or its immediately contained <closed graph reference value type>.

- 57) Every <open graph reference value type> *OGRVT* specifies the open graph reference value type with the nullability specified by *OGRVT*.
- 58) Every <closed graph reference value type> *CGRVT* specifies the closed graph reference value type whose constraining GQL-object type is the graph type identified by its immediately contained <nested graph type specification> and whose nullability is the nullability specified by *CGRVT*.
- 59) Every <binding table reference value type> *BTRVT* specifies the binding table reference value type whose constraining GQL-object type is the binding table type identified by its immediately contained <binding table type> and whose nullability is the nullability specified by *BTRVT*.
- 60) Every <node reference value type> specifies the node reference value type specified by its immediately contained <open node reference value type> or <closed node reference value type>.
- 61) Every <open node reference value type> *ONRVT* specifies the open node reference value type with the nullability specified by *ONRVT*.
- 62) Every <closed node reference value type> *CNRVT* specifies the closed node reference value type whose constraining GQL-object type is the node type identified by its immediately contained <node type specification> and whose nullability is the nullability specified by *CNRVT*.
- 63) Every <edge reference value type> specifies the edge reference value type specified by its immediately contained <open edge reference value type> or <closed edge reference value type>.
- 64) Every <open edge reference value type> *OERVT* specifies the open edge reference value type with the nullability specified by *OERVT*.
- 65) Every <closed edge reference value type> *CERVT* specifies the closed edge reference value type whose constraining GQL-object type is the edge type identified by its immediately contained <edge type specification> and whose nullability is the nullability specified by *CERVT*.
- 66) Every <constructed value type> specifies the constructed value type that is the data type specified by the <path value type>, the <list value type>, or the <record type> that it contains.
- 67) Every <path value type> *PVT* specifies the path value type with the nullability specified by *PVT*.
- 68) If GROUP is specified in the <list value type name> simply contained in <list value type> *LVT*, then the <value type> simply contained in *LVT* shall specify either a node reference value type or an edge reference value type.
- 69) Every <immaterial value type> specifies the immaterial value type that is the data type specified by the immediately contained <null type> or <empty type>
- 70) Every <null type> specifies the null type.
NOTE 289 — See Subclause 4.16.8, “Immaterial value types: null type and empty type”.
- 71) Every <empty type> specifies the empty type.
NOTE 290 — See Subclause 4.16.8, “Immaterial value types: null type and empty type”.
- 72) For every <immaterial value type> *IVT*, there is an implementation-defined (IV008) immaterial value type *IVTNF(IVT)* known as the *normal form* of *IVT* (which can be *IVT* itself), such that:
- The nullability of *IVTNF(IVT)* is the nullability of *IVT*.
 - The declared name of *IVT(IVT)* is the same as the preferred name of *IVT*.
 - All invocations of *IVTNF(IVT)* are the same.

NOTE 291 — This ensures that all invocations of the implementation-defined (IV003) choice of the normal form of an immaterial value type from amongst the equivalent immaterial value types are deterministic.

- 73) Every <list value type> *LVT* specifies a *list value type* defined as follows.
- a) Let *LET* be defined as follows.
Case:
 - i) If *LVT* simply contains a <value type> *EVT*, then *LET* is the value type specified by *EVT*.
 - ii) Otherwise, *LET* is the value type specified by *ANY*.
 - b) *LVT* specifies a *closed list value type* whose list element type is *LET*.
 - c) The group characteristics of *LVT* is defined as follows.
Case:
 - i) If *GROUP* is specified in the <list value type name> simply contained in *LVT*, then the list value type specified by *LVT* is a group list value type.
 - ii) Otherwise, the list value type specified by *LVT* is a regular list value type.
 - d) Case:
 - i) If the <max length> *LTMAXL* is specified in *LVT*, then the maximum cardinality of the list value type specified by *LVT* is the value of *LTMAXL*. The value of *LTMAXL* shall be less or equal than implementation-defined (IL015) maximum cardinality for list value types whose list element type is *LET*.
 - ii) Otherwise, the maximum cardinality of the list value type specified by *LVT* is the implementation-defined (IL015) maximum cardinality for list value types whose list element type is *LET*.
 - e) The nullability of the list value type specified by *LVT* is the nullability specified by *LVT*.
- 74) For each list value type *LVT*, there is an implementation-defined (IV008) list value type, *LNF(LVT)*, known as the *normal form* of *LVT* (which can be *LVT* itself), such that:
- a) If *LVT1* and *LVT2* are two list value types whose list element type have the same normal form, whose group characteristics are the same, whose maximum cardinalities are the same, and whose indications regarding the inclusion of the null value are the same, then $LNF(LVT1) = LNF(LVT2)$.
 - b) The group characteristic of *LNF(LVT)* is the group characteristic of *LVT*.
 - c) The nullability of *LNF(LVT)* is the nullability of *LVT*.
 - d) $LNF(LNF(LVT)) = LNF(LVT)$.
 - e) The results of all invocations of *LNF(LVT)* are the same.
- NOTE 292 — This ensures that all invocations of the implementation-defined (IV008) choice of the normal form of a list value type from amongst the equivalent exact numeric types are deterministic.
- 75) Every <record type> *RT* specifies a record type defined as follows.
- Case:
- a) If *RT* simply contains a <field types specification> *FTS*, then *RT* specifies the *closed record type* whose field types are specified by *FTS* with the nullability specified by *RT*.
 - b) Otherwise, *RT* specifies the *open record type* with the nullability specified by *RT*.

- 76) For each record type RT , there is an implementation-defined (IV008) record type, $RNF(RT)$, known as the *normal form* of RT (which can be RT itself), such that:
- a) If $RT1$ and $RT2$ are two record types that have the same set of field type names FNS , whose field type value types for each name in FNS have the same normal form, whose indications of whether the type is closed or open are the same, and whose indications regarding the inclusion of the null value are the same, then $RNF(RT1) = RNF(RT2)$.
 - b) The nullability of $RNF(RT)$ is the nullability of RT .
 - c) $RNF(RNF(RT)) = RNF(RT)$.
 - d) The results of all invocations of $RNF(RT)$ are the same.

NOTE 293 — This ensures that all invocations of the implementation-defined (IV008) choice of the normal form of a record type from amongst the equivalent exact numeric types are deterministic.

- 77) Every <field types specification> specifies the field types specified by the <field type list> that it immediately contains.
- 78) Every <field type list> specifies the field types that are specified by the <field type>s that it immediately contains.
- 79) The maximum number of <field type>s immediately contained in a <field type list> is the implementation-defined (IL015) maximum number of record fields.
- 80) The preferred name of dynamic union types is ANY.
- 81) Every <dynamic union type> specifies the dynamic union type that is the data type specified by the <open dynamic union type> or the <closed dynamic union type> that it contains.
- 82) If the <dynamic property value type> DPT is specified, then DPT is effectively replaced by a dynamic union type that specifies the dynamic property value type with the nullability specified by DPT .
- NOTE 294 — See Subclause 4.3.4, “Properties and supported property value types”, for the definitions of supported property value type and dynamic property value type.
- 83) Every <open dynamic union type> $ODUT$ specifies the *open dynamic union type* with the nullability specified by $ODUT$.
- 84) Every <closed dynamic union type> $CDUT$ specifies the *closed dynamic union type* with the component types specified by the <component type list> simply contained in $CDUT$ and with the nullability specified by $CDUT$.
- 85) The nullability specified by a <closed dynamic union type> is the nullability specified by the simply contained <component type list>.
- 86) The nullability specified by a <component type list> CTL is defined as follows. If every <value type> simply contained in CTL specifies known not nullable, then CTL specifies known not nullable; otherwise, CTL specifies possibly nullable.
- 87) If the <component type list> CTL is specified, then CTL specifies the set of value types specified by at least one of the <value type>s simply contained in CTL .
- 88) If a <component type list> CTL is specified, then either every <value type> simply contained in CTL shall specify possibly nullable or every <value type> simply contained in CTL shall specify known not nullable.
- 89) For each dynamic union type DUT , there is an implementation-defined (IV008) value type $DVNF(DUT)$ known as the *normal form* of DUT (which can be DUT itself), such that:
- a) The declared name of $DVNF(DUT)$ is the preferred name of dynamic union types.

- b) For each subtype *VT* of *DUT*: *VT* is a subtype of *DVNF(DUT)*.
- c) Every component type of *DVNF(DUT)* is a static value type.
- d) For each open value type *OVT*, the component types of *DVNF(DUT)* shall not include a proper closed subtype of *OVT*.
- e) If the component types of *DUT* include a nullable value type, then the component types of *DVNF(DUT)* include no material value types.
NOTE 295 — Otherwise, the component types of *DVNF(DUT)* include only material value types.
- f) If the component types of *DUT* include a nullable value type, then let *MDUT* be *DUT* with each component type replaced by its material variant, let *NDUT* be the dynamic union type obtained by replacing each component type of *DVNF(MDUT)* by its nullable variant and *DVNF(DUT)* is *DVNF(NDUT)*.
- g) If the component types of *DUT* include a regular list value type, then the component types of *DVNF(DUT)* include no group list value type.
- h) If the component types of *DUT* contain two value types *VT1* and *VT2* such that *VT1* is a proper supertype of *VT2* and *DVNF(DUT)* excludes *VT2*, then for every other dynamic union type *ODUT* whose component types include both *VT1* and *VT2*, it holds that *DVNF(ODUT)* also excludes *VT2*.
- i) If the component types of *DUT* are the component types of the dynamic union type specified by ANY PROPERTY VALUE, then *DVNF(DUT)* is *DVNF(ANY PROPERTY VALUE)*.
- j) If *DUT* is a closed dynamic union type, then the order in which the component types are specified by the <component type list> *CTL* simply contained in *DVNF(DUT)* is restricted as follows.
 - i) The Syntax Rules of Subclause 22.20, “Determination of value type precedence”, are applied with the set of component types of *DUT* as *NDTSET*; let *CDTLIST* be the *NDTLIST* returned from the application of those Syntax Rules.
 - ii) The component types specified by the <component type list> *CTL* simply contained in *DVNF(DUT)* shall be specified in the same order as in *CDTLIST*.
- k) If *DUT* has exactly one component type *CT*, then *DVNF(DUT)* is the <value type> that specifies *CT*.
- l) If *DVNF(DUT)* is a closed dynamic union type with component types *CTS*, then *CTS* does not include immaterial value types.
- m) *DVNF(DVNF(DUT)) = DVNF(DUT)*.
- n) The results of all invocations of *DVNF(DUT)* are the same.

NOTE 296 — This ensures that all invocations of the implementation-defined (IV008) choice of the normal form of a dynamic union type from amongst the equivalent exact numeric types are deterministic.

General Rules

- 1) If <boolean type> is specified, then a Boolean data type descriptor is created for the specified Boolean type *BT* that describes *BT* and comprises:
 - a) The declared name of the primary base type of all Boolean types (BOOLEAN DATA).
NOTE 297 — See Subclause 4.16.1, “Introduction to predefined value types and related base types”.
 - b) The preferred name of *BT*.

- c) The indication of whether *BT* includes the null value as determined by the nullability specified by the <boolean type>.
- 2) If <character string type> is specified, then a character string data type descriptor is created for the specified character string type *CST* that describes *CST* and comprises:
- a) The declared name of the primary base type of all character string types (STRING DATA).
NOTE 298 — See Subclause 4.16.1, “Introduction to predefined value types and related base types”.
 - b) The preferred name of *CST*.
 - c) The minimum length in characters of *CST*.
 - d) The maximum length in characters of *CST*.
 - e) The indication that *CST* includes the null value as determined by the nullability specified by the <character string type>.
- 3) If <byte string type> is specified, then a byte string data type descriptor is created for the specified byte string type *BST* that describes *BST* and comprises:
- a) The declared name of the primary base type of all byte string types (BINARY DATA).
NOTE 299 — See Subclause 4.16.1, “Introduction to predefined value types and related base types”.
 - b) The preferred name of *BST*.
 - c) The minimum length in bytes of *BST*.
 - d) The maximum length in bytes of *BST*.
 - e) The indication of whether *BST* includes the null value as determined by the nullability specified by the <byte string type>.
- 4) If <exact numeric type> is specified, then a numeric data type descriptor is created for the specified exact numeric type *ENT* that describes *ENT* and comprises:
- a) The declared name of the primary base type of *ENT* (EXACT NUMERIC DATA).
NOTE 300 — See Subclause 4.16.1, “Introduction to predefined value types and related base types”.
 - b) The preferred name of *ENT*, which is the declared name of the normal form of *ENT*.
 - c) The indication of whether exact numbers of *ENT* are specified in binary or decimal terms.
 - d) The precision of *ENT*.
 - e) The scale of *ENT*.
 - f) The indication of whether *ENT* includes the null value as determined by the nullability specified by the <exact numeric type>.
- 5) If <approximate numeric type> is specified, then a numeric data type descriptor is created for the specified approximate numeric type *ANT* that describes *ANT* and comprises:
- a) The declared name of the primary base type of *ANT* (FLOAT NUMERIC DATA).
NOTE 301 — See Subclause 4.16.1, “Introduction to predefined value types and related base types”.
 - b) The preferred name of *ANT*, which is the declared name of the normal form of *ANT*.
 - c) The indication that approximate numbers of *ANT* are specified in binary terms.
 - d) The precision of *ANT*.

- e) The scale of *ANT*.
 - f) The indication of whether *ANT* includes the null value as determined by the nullability specified by the <approximate numeric type>.
- 6) If <temporal instant type> is specified, then a temporal instant data type descriptor is created for the specified temporal instant type *TINT* that describes *TINT* and comprises:
- a) The declared name of the primary base type of all temporal instant types (TEMPORAL INSTANT DATA).
- NOTE 302 — See Subclause 4.16.1, “Introduction to predefined value types and related base types”.
- b) The preferred name of *TINT*, which is the declared name of the normal form of *TINT*.
 - c) The indication of whether values of *TINT* capture the date.
 - d) The indication of whether values of *TINT* capture the time of day.
 - e) The indication of whether values of *TINT* include a time zone displacement.
 - f) The indication of whether *TINT* includes the null value as determined by the nullability specified by the <temporal instant type>.
- 7) If <temporal duration type> is specified, then a temporal duration data type descriptor is created for the specified temporal duration type *TDURT* that describes *TDURT* and comprises:
- a) The declared name of the primary base type of all temporal duration types (TEMPORAL DURATION DATA).
- NOTE 303 — See Subclause 4.16.1, “Introduction to predefined value types and related base types”.
- b) The temporal duration unit group of temporal durations of *TDURT* as specified by the <temporal duration qualifier> simply contained in the <temporal duration type>.
 - c) The indication of whether *TDURT* includes the null value as determined by the nullability specified by the <temporal duration type>.
- 8) If <open graph reference value type> is specified, then a graph reference value data type descriptor is created for the specified open graph reference value type *OGRVT* that describes *OGRVT* and comprises:
- a) The reference base type name of graph reference value types (GRAPH REFERENCE).
 - b) The object base type name of graph types (GRAPH DATA).
 - c) The preferred name, which is the preferred name of graph types.
 - d) No constraining GQL-object type.
 - e) The indication of whether *OGRVT* includes the null value as determined by the nullability specified by the <open graph reference value type>.
- 9) If <closed graph reference value type> is specified, then a graph reference value data type descriptor is created for the specified graph reference value type *CGRVT* that describes *CGRVT* and comprises:
- a) The reference base type name of graph reference value types (GRAPH REFERENCE).
 - b) The object base type name of graph types (GRAPH DATA).
 - c) The preferred name, which is the preferred name of graph types.
 - d) The specified constraining GQL-object type.

- e) The indication of whether *CGRVT* includes the null value as determined by the nullability specified by the <closed graph reference value type>.
- 10) If <binding table reference value type> is specified, then a binding table reference value data type descriptor is created for the specified binding table reference value type *BTRVT* that describes *BTRVT* and comprises:
- a) The reference base type name of binding table reference value types (BINDING TABLE REFERENCE).
 - b) The object base type name of binding table types (BINDING TABLE DATA).
 - c) The preferred name, which is the preferred name of binding table types.
 - d) The specified constraining GQL-object type.
 - e) The indication of whether *BTRVT* includes the null value as determined by the nullability specified by the <binding table reference value type>.
- 11) If <open node reference value type> is specified, then a node reference value data type descriptor is created for the specified open node reference value type *ONRVT* that describes *ONRVT* and comprises:
- a) The reference base type name, which is the implementation-defined (ID090) preferred name of the base type of node reference value types (NODE REFERENCE or VERTEX REFERENCE).
 - b) The object base type name, which is the implementation-defined (ID090) preferred name of the base type of node types (NODE DATA or VERTEX DATA).
 - c) The preferred name, which is the preferred name of node types.
 - d) No constraining GQL-object type.
 - e) The indication of whether *ONRVT* includes the null value as determined by the nullability specified by the <open node reference value type>.
- 12) If <closed node reference value type> is specified, then a node reference value data type descriptor is created for the specified closed node reference value type *CNRVT* that describes *CNRVT* and comprises:
- a) The reference base type name, which is the implementation-defined (ID090) preferred name of the base type of node reference value types (NODE REFERENCE or VERTEX REFERENCE).
 - b) The object base type name, which is the implementation-defined (ID090) preferred name of the base type of node types (NODE DATA or VERTEX DATA).
 - c) The preferred name, which is the preferred name of node types.
 - d) The specified constraining GQL-object type.
 - e) The indication of whether *CNRVT* includes the null value as determined by the nullability specified by the <closed node reference value type>.
- 13) If <open edge reference value type> is specified, then an edge reference value data type descriptor is created for the specified open edge reference value type *OERV* that describes *OERV* and comprises:
- a) The reference base type name, which is the implementation-defined (ID091) preferred name of the base type of edge reference value types (EDGE REFERENCE or RELATIONSHIP REFERENCE).

- b) The object base type name, which is the implementation-defined (ID091) preferred name of the base type of edge types (EDGE DATA or RELATIONSHIP DATA).
 - c) The preferred name, which is the preferred name of edge types.
 - d) No constraining GQL-object type.
 - e) The indication of whether *OERVT* includes the null value as determined by the nullability specified by the <open edge reference value type>.
- 14) If <closed edge reference value type> is specified, then an edge reference value data type descriptor is created for the specified edge reference value type *CERVT* that describes *CERVT* and comprises:
- a) The reference base type name, which is the implementation-defined (ID091) preferred name of the base type of edge reference value types (EDGE REFERENCE or RELATIONSHIP REFERENCE).
 - b) The object base type name, which is the implementation-defined (ID091) preferred name of the base type of edge types (EDGE DATA or RELATIONSHIP DATA).
 - c) The preferred name, which is the preferred name of edge types.
 - d) The specified constraining GQL-object type.
 - e) The indication of whether *CERVT* includes the null value as determined by the nullability specified by the <closed edge reference value type>.
- 15) If <immaterial value type> is specified, then an immaterial value data type descriptor is created for the specified immaterial value type *IVT* that describes *IVT* and comprises:
- a) The declared name of the primary base type of all immaterial value types (NULL DATA).
 - b) The preferred name of *IVT*.
 - c) The indication of whether *IVT* includes the null value as determined by the nullability specified by the <immaterial value type>.
- 16) If <path value type> is specified, then let *PVT* be the specified path value type and a path value data type descriptor is created for *PVT* that describes *PVT* and comprises:
- a) The declared name of the primary base type of all path value types (PATH VALUE DATA).
NOTE 304 — See Subclause 4.15.1, “Introduction to constructed value types and related base types”.
 - b) The indication of whether *PVT* includes the null value as determined by the nullability specified by the <path value type>.
- 17) If <list value type> is specified, then a list value data type descriptor is created for *LVT* that describes *LVT* and comprises:
- a) The base type name of all list value types (LIST VALUE DATA).
 - b) The preferred name *LNF(LVT)*, which is the declared name of the normal form of the list value type.
 - c) The specified list element type (*LET*).
 - d) The specified group characteristic of the list value type.
 - e) The specified maximum cardinality of the list value type.
 - f) The indication of whether *LVT* includes the null value as determined by the nullability specified by the <list value type>.

- 18) If the specified list element type of a list value type *LVT* is an open dynamic union type, then *LVT* is open; otherwise, *LVT* is closed.
- 19) If <record type> is specified, then let *RT* be the specified record type.

Case:

- a) If *RT* is a closed record type, then a record data type descriptor is created for *RT* that describes *RT* and comprises:
- i) The base type name of all record types (RECORD DATA).
 - ii) The indication that *RT* is closed.
 - iii) A field type descriptor for every <field type> simply contained in the <record type>, according to the Syntax Rules and General Rules of Subclause 18.10, "<field type>", applied to the <field type>s in the order in which they were specified.
 - iv) The indication of whether *RT* includes the null value as determined by the nullability specified by the <record type>.
- b) Otherwise, *RT* is the open record type and a record data type descriptor is created for *RT* that describes *RT* and comprises:
- i) The base type name of all record types (RECORD DATA).
 - ii) The indication that *RT* is open.
 - iii) The indication of whether *RT* includes the null value as determined by the nullability specified by the <record type>.
- 20) If <dynamic union type> is specified, then let *DUT* be the specified dynamic union type and a dynamic union data type descriptor is created for *DUT* that describes *DUT* and comprises:
- a) The declared name of the primary base type of all dynamic union types, which is the dynamic base type of all value types (ANY DATA).
- NOTE 305 — See Subclause 4.14.1, "Introduction to dynamic union types and the dynamic base type".
- b) The preferred name of *DUT*, which is the declared name of the normal form of *DUT*.
- c) The indication of whether *DUT* is open or closed as determined by the <dynamic union type>.
- d) The component types of *DUT* are defined as follows.

Case:

- i) If *DUT* specifies a closed dynamic union type, then the component types of *DUT* are given by the set of all data types specified by the <component type list> simply contained in the <dynamic union type>.
- ii) Otherwise, *DUT* specifies an open dynamic union type.
- 1) Let *COMPTYPES* be an implementation-defined (IV014) set of nullable value types that includes at least one supertype of every static value type supported by the GQL-implementation.
 - 2) If the <dynamic union type> specifies possibly nullable, then the component types of *DUT* are *COMPTYPES*; otherwise, the component types of *DUT* are given by the set of all material variants of value types from *COMPTYPES*.

- e) The indication of whether *DUT* includes the null value as determined by the nullability specified by the <dynamic union type>.
- 21) A dynamic union type *DUT* includes every material value of one of its component types *DUT*. If *DUT* is nullable, then it additionally includes the null value. No other values are included in *DUT*.

Conformance Rules

- 1) Conforming GQL language shall not specify a <list value type name> that contains GROUP.
NOTE 306 — A <list value type name> that contains GROUP is a specification device to indicate the group characteristic of a list value type in the syntactic representation of the type and is not syntax available to the user.
- 2) Without Feature GV02, “8 bit signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains SIGNED INTEGER8, INTEGER8, or INT8.
- 3) Without Feature GV18, “Small signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains SMALLINT.
- 4) Without Feature GV19, “Big signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains BIGINT.
- 5) Without Feature GV04, “16 bit signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains SIGNED INTEGER16, INTEGER16, or INT16.
- 6) Without Feature GV07, “32 bit signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains SIGNED INTEGER32, INTEGER32, or INT32.
- 7) Without Feature GV12, “64 bit signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains SIGNED INTEGER64, INTEGER64, or INT64.
- 8) Without Feature GV14, “128 bit signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains INT128, INTEGER128, or SIGNED INTEGER128.
- 9) Without Feature GV16, “256 bit signed integer numbers”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains INT256, INTEGER256, or SIGNED INTEGER256.
- 10) Without Feature GV01, “8 bit unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UNSIGNED INTEGER8 or UINT8.
- 11) Without Feature GV03, “16 bit unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UNSIGNED INTEGER16 or UINT16.
- 12) Without Feature GV06, “32 bit unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UNSIGNED INTEGER32 or UINT32.
- 13) Without Feature GV11, “64 bit unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UNSIGNED INTEGER64 or UINT64.
- 14) Without Feature GV05, “Small unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains USMALLINT.
- 15) Without Feature GV08, “Regular unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UINT.
- 16) Without Feature GV10, “Big unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UBIGINT.
- 17) Without Feature GV09, “Specified integer number precision”, conforming GQL language shall not contain a <signed binary exact numeric type> that contains a <precision> or a <scale>.

ISO/IEC 39075:2024(en)
18.9 <value type>

- 18) Without Feature GV13, “128 bit unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UINT128, or UNSIGNED INTEGER128.
- 19) Without Feature GV15, “256 bit unsigned integer numbers”, conforming GQL language shall not contain an <unsigned binary exact numeric type> that contains UINT256, or UNSIGNED INTEGER256.
- 20) Without Feature GV20, “16 bit floating point numbers”, conforming GQL language shall not contain an <approximate numeric type> that contains FLOAT16.
- 21) Without Feature GV21, “32 bit floating point numbers”, conforming GQL language shall not contain an <approximate numeric type> that contains FLOAT32.
- 22) Without Feature GV24, “64 bit floating point numbers”, conforming GQL language shall not contain an <approximate numeric type> that contains FLOAT64.
- 23) Without Feature GV25, “128 bit floating point numbers”, conforming GQL language shall not contain an <approximate numeric type> that contains FLOAT128.
- 24) Without Feature GV26, “256 bit floating point numbers”, conforming GQL language shall not contain an <approximate numeric type> that contains FLOAT256.
- 25) Without Feature GV23, “Floating point type name synonyms”, conforming GQL language shall not contain an <approximate numeric type> that contains REAL or DOUBLE.
- 26) Without Feature GV22, “Specified floating point number precision”, conforming GQL language shall not contain an <approximate numeric type> that contains a <precision> or a <scale>.
- 27) Without Feature GV17, “Decimal numbers”, conforming GQL language shall not contain an <exact numeric type> that contains DECIMAL or DEC.
- 28) Without Feature GV35, “Byte string types”, conforming GQL language shall not contain a <byte string type>.
- 29) Without Feature GV36, “Specified byte string minimum length”, conforming GQL language shall not contain a <byte string type> that contains a <min length>.
- 30) Without Feature GV37, “Specified byte string maximum length”, conforming GQL language shall not contain a <byte string type> that contains a <max length>.
- 31) Without Feature GV38, “Specified byte string fixed length”, conforming GQL language shall not contain a <byte string type> that contains a <fixed length>.
- 32) Without Feature GV30, “Specified character string minimum length”, conforming GQL language shall not contain a <character string type> that contains a <min length>.
- 33) Without Feature GV31, “Specified character string maximum length”, conforming GQL language shall not contain a <character string type> that contains a <max length>.
- 34) Without Feature GV32, “Specified character string fixed length”, conforming GQL language shall not contain a <character string type> that contains a <fixed length>.
- 35) Without Feature GV39, “Temporal types: date, local datetime and local time support”, conforming GQL language shall not contain a <temporal instant type> that is a <localdatetime type>, a <date type>, or a <localtime type>.
- 36) Without Feature GV40, “Temporal types: zoned datetime and zoned time support”, conforming GQL language shall not contain a <temporal instant type> that is a <datetime type> or a <time type>.
- 37) Without Feature GV41, “Temporal types: duration support”, conforming GQL language shall not contain a <temporal duration type>.

- 38) Without Feature GV71, “Immaterial value types: null type support”, conforming GQL language shall not contain a <null type>.
- 39) Without Feature GV72, “Immaterial value types: empty type support”, conforming GQL language shall not contain an <empty type>.
- 40) Without Feature GV50, “List value types”, conforming GQL language shall not contain a <list value type>.
- 41) Without Feature GV45, “Record types”, conforming GQL language shall not contain a <record type>.
- 42) Without Feature GV46, “Closed record types”, conforming GQL language shall not contain a <record type> containing a <field types specification>.
- 43) Without Feature GV47, “Open record types”, conforming GQL language shall not contain a <record type> not containing a <field types specification>.
- 44) Without Feature GV55, “Path value types”, conforming GQL language shall not contain a <path value type>.
- 45) Without Feature GV60, “Graph reference value types”, conforming GQL language shall not contain a <graph reference value type>.
- 46) Without Feature GV61, “Binding table reference value types”, conforming GQL language shall not contain a <binding table reference value type>.
- 47) Without Feature GV66, “Open dynamic union types”, conforming GQL language shall not contain an <open dynamic union type> or a <list value type> that does not simply contain a <value type>.
- 48) Without Feature GV67, “Closed dynamic union types”, conforming GQL language shall not contain a <closed dynamic union type>.
- 49) Without Feature GV68, “Dynamic property value types”, conforming GQL language shall not contain a <dynamic property value type>.
- 50) Without Feature GV90, “Explicit value type nullability”, conforming GQL language shall not contain <not null>.
- 51) Conforming GQL language shall not contain a <dynamic union type> that simply contains exactly one <component type>

18.10 <field type>

Function

Specify a field type.

Format

```
<field type> ::=  
  <field name> [ <typed> ] <value type>
```

Syntax Rules

- 1) Let *FTL* be the <field type list> that simply contains the <field type> *FT*.
- 2) The <field name> shall not be equivalent to the <field name> of any other <field type> simply contained in *FTL*.
- 3) The *name* of *FT* is the name specified by the <field name> simply contained in *FT*.
- 4) The *value type* of *FT* is the value type specified by the <value type> simply contained in *FT*.

General Rules

- 1) A data type descriptor is created that describes the value type of the field type being defined.
- 2) A field type descriptor is created that describes the field type being defined. The field type descriptor includes the following:
 - a) The name of the field type.
 - b) The value type of the field type.

Conformance Rules

- 1) Without Feature GV48, “Nested record types”, conforming GQL language shall not contain a <field type> that simply contains a <record type>.

19 Predicates

19.1 <search condition>

Function

Specify a condition that is *True*, *False*, or *Unknown*, depending on the result of a <boolean value expression>.

Format

<search condition> ::=
 <boolean value expression>

Syntax Rules

None.

General Rules

- 1) The result of the <search condition> is the result of the <boolean value expression>.
- 2) A <search condition> is said to be *satisfied* if and only if the result of the <boolean value expression> is *True*.

Conformance Rules

None.

19.2 <predicate>

Function

Specify a condition that can be evaluated to give a Boolean value.

Format

```
<predicate> ::=  
    <comparison predicate>  
    | <exists predicate>  
    | <null predicate>  
    | <normalized predicate>  
    | <value type predicate>  
    | <directed predicate>  
    | <labeled predicate>  
    | <source/destination predicate>  
    | <all_different predicate>  
    | <same predicate>  
    | <property_exists predicate>
```

Syntax Rules

None.

General Rules

- 1) The result of a <predicate> is the result of the immediately contained <comparison predicate>, <exists predicate>, <null predicate>, <value type predicate>, <normalized predicate>, <directed predicate>, <labeled predicate>, <source/destination predicate>, <all_different predicate>, <same predicate>, or <property_exists predicate>.

Conformance Rules

None.

19.3 <comparison predicate>

Function

Specify a comparison of two values.

Format

```
<comparison predicate> ::=  
  <comparison predicand> <comparison predicate part 2>  
  
<comparison predicate part 2> ::=  
  <comp op> <comparison predicand>  
  
<comp op> ::=  
  <equals operator>  
  | <not equals operator>  
  | <less than operator>  
  | <greater than operator>  
  | <less than or equals operator>  
  | <greater than or equals operator>  
  
<comparison predicand> ::=  
  <common value expression>  
  | <boolean predicand>
```

Syntax Rules

- 1) Let L and R respectively denote the first and second <comparison predicand>s.
- 2) Case:
 - a) If <comp op> is <equals operator> or <not equals operator>, then L and R are operands of an equality operation. The Syntax Rules and Conformance Rules of [Subclause 22.13, “Equality operations”](#), apply.
 - b) Otherwise, L and R are operands of an ordering operation. The Syntax Rules and Conformance Rules of [Subclause 22.14, “Ordering operations”](#), apply.
- 3) Case:
 - a) If the declared types of L and R are list value types with list element types LET and RET , respectively, then let LV and RV be <value expression>s whose declared types are LET and RET , respectively. The Syntax Rules of this Subclause are applied to:
$$LV \text{ <comp op> } RV$$
 - b) If the declared types of L and R are record types with identical sets of field type names FNS , then let N be the cardinality of FNS , let FN_i , $1 \text{ (one)} \leq i \leq N$, be the i -th name in a permutation of FNS , let LFV_i denote some <value expression> whose declared type is the value type of the field type with name FN_i of the declared types of L , and let RFV_i denote some <value expression> whose declared type is the value type of the field type with name FN_i of the declared types of R . For i , $1 \text{ (one)} \leq i \leq N$, the Syntax Rules of this Subclause are applied to:
$$LFV_i \text{ <comp op> } RFV_i$$
- 4) Let CP be the <comparison predicate>. The following syntactic transformations are applied.

ISO/IEC 39075:2024(en)
19.3 <comparison predicate>

Case:

- a) If the <comp op> is <not equals operator>, then *CP* is equivalent to:

NOT ($L = R$)

- b) If the <comp op> is <greater than operator>, then *CP* is equivalent to:

($R < L$)

- c) If the <comp op> is <less than or equals operator>, then *CP* is equivalent to:

($L < R$
OR
 $L = R$)

- d) If the <comp op> is <greater than or equals operator>, then *CP* is equivalent to:

($R < L$
OR
 $R = L$)

General Rules

- 1) Let *LV* and *RV* be the results of the <comparison predicate>s *L* and *R*, respectively.
- 2) The result of *CP* is defined as follows.

Case:

- a) If *LV* and *RV* are universally comparable values, then

Case:

- i) If the <comp op> is <equals operator>, then the result of *CP* is False.

NOTE 307 — This ensures that every two values that are defined in this document as not essentially comparable values are never considered equal.

- ii) Otherwise, the <comp op> is <less than operator> and the result of *CP* is determined according to an implementation-defined (IV010) total order. This total order shall be stable during the execution of the currently executing GQL-request.

- b) If *LV* and *RV* are essentially comparable values, then:

Case:

- i) If at least one of *LV* and *RV* is the null value, then the result of *CP* is Unknown.

- ii) If *LV* and *RV* are list values, then:

- 1) Let *NL* and *NR* be the cardinalities of *LV* and *RV*, respectively.
- 2) Let *MINL* be the minimum of *NL* and *NR*.
- 3) For *i*, 1 (one) ≤ *i* ≤ *NL*: Let *LE_i* denote a <value expression> whose value and declared type is that of the *i*-th element of *LV*.
- 4) For *i*, 1 (one) ≤ *i* ≤ *NR*: Let *RE_i* denote a <value expression> whose value and declared type is that of the *i*-th element of *RV*.
- 5) Case:

ISO/IEC 39075:2024(en)
19.3 <comparison predicate>

- A) If the <comp op> is <equals operator>, then
Case:
I) If $NL = 0$ (zero) and $NR = 0$ (zero), then the result of CP is True.
II) If $NL = NR$ and $LE_i = RE_i$ is True for all i , 1 (one) $\leq i \leq NL = NR$, then the result of CP is True.
III) If $NL \neq NR$ or NOT ($LE_i = RE_i$) is True for some i , 1 (one) $\leq i \leq MINL$ then the result of CP is False.
IV) Otherwise, the result of CP is Unknown.
- B) Otherwise, the <comp op> is <less than operator> and the result of CP is implementation-defined (IV002).
- iii) If LV and RV are records, then:
1) LV and RV have the same set of field names FNS .
2) Let N be the cardinality of FNS .
3) For i , 1 (one) $\leq i \leq N$: Let FN_i be the i -th field name in a permutation of FNS , let LFV_i denote a <value expression> whose value is the value of the field with name FN_i of LV and whose declared type is the value type of the field type with name FN_i of the declared type of LV , and let RFV_i denote a <value expression> whose value is the value of the field with name FN_i of RV and whose declared type is the value type of the field type with name FN_i of the declared type of RV .
4) Case:
A) If the <comp op> is <equals operator>, then
Case:
I) If $LFV_i = RFV_i$ is True for all i , 1 (one) $\leq i \leq N$ then the result of CP is True.
II) If NOT ($LFV_i = RFV_i$) is True for some i , 1 (one) $\leq i \leq N$ then the result of CP is False.
III) Otherwise, the result of CP is Unknown.
B) Otherwise, the <comp op> is <less than operator> and the result of CP is implementation-defined (IV002).
- iv) If LV and RV are path values, then:
1) Let $LPEL$ and $RPEL$ be the path element lists of LV and RV , respectively.
2) Case:
A) If the <comp op> is <equals operator>, then
Case:
I) If $LPEL = RPEL$ is True, then the result of CP is True.
II) Otherwise, the result of CP is False.
B) Otherwise, the <comp op> is <less than operator> and

ISO/IEC 39075:2024(en)
19.3 <comparison predicate>

Case:

- I) If $LPEL < RPEL$ is True, then the result of CP is True.
- II) Otherwise, the result of CP is False.

v) Otherwise, LV and RV are values of predefined value types and:

1) If the <comp op> is <equals operator>, then:

A) If LV and RV are equal, then the result of CP is True.

NOTE 308 — Whether LV is equal to RV is determined by an application of [General Rule 3](#) of this Subclause, which defines the comparison of essentially comparable values of predefined value types.

B) Otherwise, the result of CP is False.

2) If the <comp op> is <less than operator>, then:

A) If LV is less than RV , then the result of CP is True.

NOTE 309 — Whether LV is less than RV is determined by an application of [General Rule 3](#) of this Subclause, which defines the comparison of essentially comparable values of predefined value types.

B) Otherwise, the result of CP is False.

c) Otherwise, LV and RV are not comparable values and an exception condition is raised: *data exception — values not comparable (22G04)*.

3) Given two essentially comparable values $V1$ and $V2$ of predefined value types, the comparison of $V1$ and $V2$, i.e., whether $V1$ is less than, equal to, or greater than $V2$, is defined as follows:

a) If $V1$ and $V2$ are numbers, then they are compared with respect to their numerical values.

b) If $V1$ and $V2$ are character strings, then:

i) If the length in characters of $V1$ is not equal to the length in characters of $V2$, then it is implementation-defined ([IA015](#)) if the shorter character string is effectively replaced, for the purposes of comparison, with a copy of itself that has been extended to the length of the longer character string by concatenation on the right of one or more <space> characters.

ii) The Syntax Rules of [Subclause 22.16, “Determination of collation”](#), are applied; let CS be the *COLL* returned from the application of those Syntax Rules.

iii) The result of the comparison of $V1$ and $V2$ is given by the collation CS .

c) If $V1$ and $V2$ are byte strings, then:

i) Let $LENGTH\_V1$ be the length in bytes of $V1$ and let $LENGTH\_V2$ be the length in bytes of $V2$. For i , $1 \text{ (one)} \leq i \leq LENGTH\_V1$, let $V1_i$ be the i -th byte of $V1$, and For i , $1 \text{ (one)} \leq i \leq LENGTH\_V2$, let $V2_i$ be the i -th byte of $V2$.

ii) If $LENGTH\_V1 = LENGTH\_V2$ and for i , $1 \text{ (one)} \leq i \leq LENGTH\_V1$, $V1_i = V2_i$, then $V1$ is equal to $V2$.

iii) If $LENGTH\_V1 < LENGTH\_V2$, $V1_i = V2_i$ for all $i \leq LENGTH\_V1$, and the right-most $LENGTH\_V2 - LENGTH\_V1$ bytes of $V2$ are all X'00's, then it is implementation-defined ([IA016](#)) whether $V1$ is equal to $V2$ or whether $V1$ is less than $V2$.

ISO/IEC 39075:2024(en)
19.3 <comparison predicate>

- iv) If $LENGTH\_V1 < LENGTH\_V2$, $V1_i = V2_i$ for all $i \leq LENGTH\_V1$, and at least one of the right-most $LENGTH\_V2 - LENGTH\_V1$ bytes of $V2$ is not X'00's, then $V1$ is less than $V2$.
- v) If $V1_j < V2_j$, for some j , 0 (zero) $< j \leq \text{minimum}(LENGTH\_V1, LENGTH\_V2)$, and $V1_i = V2_i$ for all $i < j$, then $V1$ is less than $V2$.
- vi) Otherwise, $V1$ is greater than $V2$.
- d) If $V1$ and $V2$ are temporal instants, then:
 - i) Let RD be the duration that results from $DURATION\_BETWEEN(V1, V2)$.
 - ii) Case:
 - 1) If RD is zero, then $V1$ is equal to $V2$.
 - 2) If RD is negative, then $V1$ is less than $V2$.
 - 3) Otherwise, RD is positive and $V1$ is greater than $V2$.
- e) If $V1$ and $V2$ are year and month-based durations, then they are compared with respect to their corresponding values after conversion to a duration expressed in months.
- f) If $V1$ and $V2$ are day and time-based durations, then they are compared with respect to their corresponding values after conversion to a duration expressed in seconds.
- g) If $V1$ and $V2$ are Boolean values, then
Case:
 - i) If $V1$ and $V2$ are either both False or both True, then $V1$ and $V2$ are equal.
 - ii) If $V1$ is False and $V2$ is True, then $V1$ is less than $V2$.
 - iii) Otherwise, $V1$ is True, $V2$ is False, and $V1$ is greater than $V2$.
- h) Otherwise, $V1$ and $V2$ are reference values of the same static base type, and:
Case:
 - i) If $V1$ and $V2$ reference the same referent, then $V1$ is equal to $V2$.
 - ii) Otherwise, $V1$ and $V2$ reference different referents and then:
 - 1) $V1$ is not equal to $V2$.
 - 2) Whether $V1$ is less than or greater than $V2$ is implementation-defined (IV002).

Conformance Rules

None.

19.4 <exists predicate>

Function

Specify an existential subquery.

Format

```
<exists predicate> ::=  
  EXISTS {  
    <left brace> <graph pattern> <right brace>  
  | <left paren> <graph pattern> <right paren>  
  | <left brace> <match statement block> <right brace>  
  | <left paren> <match statement block> <right paren>  
  | <nested query specification>  
  }
```

Syntax Rules

- 1) Let *EP* be the <exists predicate>.
- 2) If *EP* immediately contains the <graph pattern> *GP*, then:
 - a) Let *RIA* be a new system-generated identifier.
 - b) *EP* is effectively replaced by:

```
EXISTS { MATCH GP RETURN TRUE AS RIA }
```
- 3) If *EP* immediately contains the <match statement block> *MSB*, then:
 - a) Let *RIA2* be a new system-generated identifier.
 - b) *EP* is effectively replaced by:

```
EXISTS { MSB RETURN TRUE AS RIA2 }
```
- 4) Let *NQS* be the <nested query specification> immediately contained in *EP*.
- 5) The declared type of *NQS* shall be a binding table type.

General Rules

- 1) In a new child execution context:
 - a) The General Rules of *NQS* are applied.
 - b) If the current execution result is a non-empty table, then the result of *EP* is True; otherwise, the result of *EP* is False.

Conformance Rules

- 1) Without Feature GQ22, “EXISTS predicate: multiple MATCH statements”, in conforming GQL language, an <exists predicate> shall not directly contain a <match statement block>

19.5 <null predicate>

Function

Specify a test for a null value.

Format

```
<null predicate> ::=  
  <value expression primary> <null predicate part 2>  
  
<null predicate part 2> ::=  
  IS [ NOT ] NULL
```

Syntax Rules

None.

General Rules

- 1) Let *VEP* be the <value expression primary>.
- 2) Let *VEPR* be the result of *VEP*.
- 3) Case:
 - a) If *VEPR* is the null value, then “*VEP* IS NULL” is True and the result of “*VEP* IS NOT NULL” is False.
 - b) Otherwise, “*VEP* IS NULL” is False and the result of “*VEP* IS NOT NULL” is True.

Conformance Rules

None.

19.6 <value type predicate>

Function

Specify a value type test.

Format

```
<value type predicate> ::=  
  <value expression primary> <value type predicate part 2>  
  
<value type predicate part 2> ::=  
  IS [ NOT ] <typed> <value type>
```

Syntax Rules

- 1) Let *VTP* be the <value type predicate>.
- 2) Let *VE* be the <value expression primary> simply contained in *VTP*, and let *VT* be the <value type> simply contained in *VTP*.
- 3) The declared type of *VTP* is the Boolean type.

General Rules

- 1) Let *VALUE* be the result of *VE* and let *TYPE* be the value type specified by *VT*.
- 2) The result of *VTP* is defined as follows.
Case:
 - a) If *VALUE* is of *TYPE* and NOT is not specified, then the result of *VTP* is True.
 - b) If *VALUE* is not of *TYPE* and NOT is specified, then the result of *VTP* is True.
 - c) Otherwise, the result of *VTP* is False.

Conformance Rules

- 1) Without Feature GA06, “Value type predicate”, conforming GQL language shall not contain a <value type predicate> or a <value type predicate part 2>.

19.7 <normalized predicate>

Function

Determine whether a character string value is normalized.

Format

```
<normalized predicate> ::=  
  <string value expression> <normalized predicate part 2>  
  
<normalized predicate part 2> ::=  
  IS [ NOT ] [ <normal form> ] NORMALIZED
```

Syntax Rules

- 1) Let <string value expression> be *SVE*.
- 2) Case:
 - a) If <normal form> is specified, then let *NF* be <normal form>.
 - b) Otherwise, let *NF* be NFC.
- 3) The expression
SVE IS NOT *NF* NORMALIZED
is equivalent to:
NOT (*SVE* IS *NF* NORMALIZED)

General Rules

- 1) The result of *SVE* IS *NF* NORMALIZED is
Case:
 - a) If the result of *SVE* is the null value, then Unknown.
 - b) If the result of *SVE* is in the normalization form specified by *NF*, in accordance with [Unicode Standard Annex #15](#), then True.
 - c) Otherwise, False.

Conformance Rules

None.

19.8 <directed predicate>

Function

Determine whether an edge variable is bound to a directed edge.

Format

```
<directed predicate> ::=  
  <element variable reference> <directed predicate part 2>  
  
<directed predicate part 2> ::=  
  IS [ NOT ] DIRECTED
```

Syntax Rules

- 1) Let *DP* be the <directed predicate>.
- 2) Let *EVR* be the <element variable reference> simply contained in *DP*. *EVR* shall have singleton degree of reference.
- 3) The declared type of *EVR* shall be an edge reference value type.
- 4) If NOT is specified, then the <directed predicate> is equivalent to:

NOT (*EVR* IS DIRECTED)

General Rules

- 1) Let *GE* be the result of *EVR*.
- 2) The result of *DP* is defined as follows.
Case:
 - a) If *GE* is the null value, then the result of *DP* is Unknown.
 - b) If *GE* is a reference value whose referent is a directed edge, then the result of *DP* is True.
 - c) Otherwise, *GE* is a reference value whose referent is an undirected edge and the result of *DP* is False.

Conformance Rules

- 1) Without Feature G110, "IS DIRECTED predicate", conforming GQL language shall not contain a <directed predicate> or a <directed predicate part 2>.

19.9 <labeled predicate>

Function

Determine whether a graph element satisfies a <label expression>.

Format

```
<labeled predicate> ::=  
  <element variable reference> <labeled predicate part 2>  
  
<labeled predicate part 2> ::=  
  <is labeled or colon> <label expression>  
  
<is labeled or colon> ::=  
  IS [ NOT ] LABELED  
  | <colon>
```

Syntax Rules

- 1) Let *LP* be the <labeled predicate>.
- 2) Let *EVR* be the <element variable reference> simply contained in *LP*. *EVR* shall have singleton degree of reference.
- 3) Let *LE* be the <label expression> simply contained in *LP*.
- 4) If NOT is specified, then *LP* is effectively replaced by:

NOT (*EVR* IS LABELED *LE*)
- 5) If *LP* simply contains an <is labeled or colon> *ILOC* that is <colon>, then *ILOC* is effectively replaced by IS LABELED.

General Rules

- 1) Let *GE* be the result of *EVR*.
- 2) The result of *LP* is defined as follows.
Case:
 - a) If *GE* is the null value, then the result of *LP* is Unknown.
 - b) Otherwise, *GE* is a graph element reference value and:
 - i) The Syntax Rules of Subclause 22.5, “Satisfaction of a <label expression> by a label set”, are applied with *LE* as *LABEL EXPRESSION* and the label set of *GE* as *LABEL SET*; let *TV* be the *TRUTH VALUE* returned from the application of those Syntax Rules.
 - ii) The result of *LP* is *TV*.

Conformance Rules

- 1) Without Feature G111, “IS LABELED predicate”, conforming GQL language shall not contain a <labeled predicate> or a <labeled predicate part 2>.

19.10 <source/destination predicate>

Function

Determine whether a node is the source or destination of an edge.

Format

```
<source/destination predicate> ::=  
    <node reference> <source predicate part 2>  
    | <node reference> <destination predicate part 2>  
  
<node reference> ::=  
    <element variable reference>  
  
<source predicate part 2> ::=  
    IS [ NOT ] SOURCE OF <edge reference>  
  
<destination predicate part 2> ::=  
    IS [ NOT ] DESTINATION OF <edge reference>  
  
<edge reference> ::=  
    <element variable reference>
```

Syntax Rules

- 1) Let *SDP* be the <source/destination predicate>.
- 2) Let *NR* be the <element variable reference> immediately contained in the <node reference> simply contained in *SDP*. *NR* shall have singleton degree of reference. The declared type of *NR* shall be a node reference value type.
- 3) Let *ER* be the <element variable reference> immediately contained in the <edge reference> simply contained in *SDP*. *ER* shall have singleton degree of reference. The declared type of *ER* shall be an edge reference value type.
- 4) Let *SOD* be the keyword SOURCE or DESTINATION simply contained in the <source/destination predicate>.
- 5) If NOT is specified, then the <source/destination predicate> is equivalent to:

NOT (*NR* IS *SOD* OF *ER*)

General Rules

- 1) Let *GEN* be the result of *NR* and let *GEE* be the result of *ER*.
- 2) The result of *SDP* is defined as follows.
Case:
 - a) If *GEN* is the null value or *GEE* is the null value, then the result of *SDP* is Unknown.
 - b) If the referent of *GEE* is an undirected edge, then the result of *SDP* is False.
 - c) If *SOD* is SOURCE and the referent of *GEN* is the source node of the referent of *GEE*, then the result of *SDP* is True.

ISO/IEC 39075:2024(en)
19.10 <source/destination predicate>

- d) If *SOD* is DESTINATION and the referent of *GEN* is the destination node of the referent of *GEE*, then the result of *SDP* is True.
- e) Otherwise, the result of *SDP* is False.

Conformance Rules

- 1) Without Feature G112, “IS SOURCE and IS DESTINATION predicate”, conforming GQL language shall not contain a <source/destination predicate>, a <source predicate part 2>, or a <destination predicate part 2>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

19.11 <all_different predicate>

Function

Determine whether all graph elements bound to a list of element references are pairwise different from one another.

Format

```
<all_different predicate> ::=  
  ALL_DIFFERENT <left paren>  
    <element variable reference> <comma> <element variable reference>  
    [ { <comma> <element variable reference> }... ]  
    <right paren>
```

Syntax Rules

- 1) Let *ADP* be the <all_different predicate>.
- 2) Every <element variable reference> simply contained in *ADP* shall have singleton degree of reference.

General Rules

- 1) Let *N* be the number of <element variable reference>s simply contained in *ADP*, let *EVR*₁, ..., *EVR*_{*N*} be an enumeration of those <element variable reference>s, and for every *i*, 1 (one) ≤ *i* ≤ *N*, let *GE*_{*i*} be the result of *EVR*_{*i*}.
- 2) The result of *ADP* is defined as follows:
Case:
 - a) If there exists *i*, 1 (one) ≤ *i* ≤ *N*, such that *GE*_{*i*} is the null value, then an exception condition is raised: *data exception — null value not allowed (22004)*.
 - b) If there exists *i*, 1 (one) ≤ *i* < *N*, and *j*, *i* < *j* ≤ *N* such that *GE*_{*i*} and *GE*_{*j*} are not comparable values, then an exception condition is raised: *data exception — values not comparable (22G04)*.
 - c) If there exist *j* and *k*, 1 (one) ≤ *j* < *k* ≤ *N*, such that *GE*_{*j*} and *GE*_{*k*} reference the same graph element, then the result of *ADP* is *False*.
 - d) Otherwise, the result of *ADP* is *True*.

Conformance Rules

- 1) Without Feature G113, “ALL_DIFFERENT predicate”, conforming GQL language shall not contain an <all_different predicate>.

19.12 <same predicate>

Function

Determine whether all element references in a list of element references bind to the same graph element.

Format

```
<same predicate> ::=  
  SAME <left paren>  
    <element variable reference> <comma> <element variable reference>  
    [ { <comma> <element variable reference> }... ]  
  <right paren>
```

Syntax Rules

- 1) Let *SDP* be the <same predicate>.
- 2) Every <element variable reference> simply contained in *SDP* shall have singleton degree of reference.

General Rules

- 1) Let *N* be the number of <element variable reference>s simply contained in *SDP*, let *EVR*₁, ..., *EVR*_{*N*} be an enumeration of those <element variable reference>s, and for every *i*, 1 (one) ≤ *i* ≤ *N*, let *GE*_{*i*} be the result of *EVR*_{*i*}.
- 2) The result of *SDP* is defined as follows.
Case:
 - a) If there exists *i*, 1 (one) ≤ *i* ≤ *N*, such that *GE*_{*i*} is the null value, then an exception condition is raised: *data exception — null value not allowed (22004)*.
 - b) If there exists *i*, 1 (one) ≤ *i* < *N*, and *j*, *i* < *j* ≤ *N* such that *GE*_{*i*} and *GE*_{*j*} are not comparable values, then an exception condition is raised: *data exception — values not comparable (22G04)*.
 - c) If every *GE*_{*i*}, 1 (one) ≤ *i* ≤ *N*, references the same graph element, then the result of *SDP* is True.
 - d) Otherwise, the result of *SDP* is False.

Conformance Rules

- 1) Without Feature G114, “SAME predicate”, conforming GQL language shall not contain a <same predicate>.

19.13 <property_exists predicate>

Function

Determine if a referenced graph element has a property.

Format

```
<property_exists predicate> ::=  
  PROPERTY_EXISTS  
    <left paren> <element variable reference> <comma> <property name> <right paren>
```

Syntax Rules

- 1) Let *PEP* be the <property_exists predicate>.
- 2) Let *EVR* be the <element variable reference> simply contained in *PEP*. *EVR* shall have singleton degree of reference.
- 3) Let *PN* be the name specified by the <property name> simply contained in *PEP*.

General Rules

- 1) Let *GRV* be the result of *EVR*.
- 2) The result of *PEP* is defined as follows.
Case:
 - a) If *GRV* is the null value, then the result of *PEP* is Unknown.
 - b) If the referent of *GRV* has a property whose name is *PN*, then the result of *PEP* is True.
 - c) Otherwise, the result of *PEP* is False.

Conformance Rules

- 1) Without Feature G115, "PROPERTY_EXISTS predicate", conforming GQL language shall not contain a <property_exists predicate>.

20 Value expressions and specifications

20.1 <value expression>

Function

Specify a constant or a value.

Format

```

<value expression> ::=
    <common value expression>
  | <boolean value expression>

<common value expression> ::=
    <numeric value expression>
  | <string value expression>
  | <datetime value expression>
  | <duration value expression>
  | <list value expression>
  | <record expression>
  | <path value expression>
  | <reference value expression>

<reference value expression> ::=
    <graph reference value expression>
  | <binding table reference value expression>
  | <node reference value expression>
  | <edge reference value expression>

<graph reference value expression> ::=
    [ PROPERTY ] GRAPH <graph expression>
  | <value expression primary>

<binding table reference value expression> ::=
    [ BINDING ] TABLE <binding table expression>
  | <value expression primary>

<node reference value expression> ::=
    <value expression primary>

<edge reference value expression> ::=
    <value expression primary>

<record expression> ::=
    <value expression primary>

<aggregating value expression> ::=
    <value expression>

```

Syntax Rules

- 1) If a <value expression> is specified that is not a <return item>, then the declared type of the <value expression> shall not be the empty type.

NOTE 310 — A record type with a field type whose value type is the empty type is unconstructable, i.e., comprises no records. Hence, a binding table whose declared type is a binding table type with such a record type has no records. The exception regarding <return item>s stated by this Syntax Rules enables renaming of columns of such binding tables since iteration is guaranteed to not occur in a <select statement> or a <return statement> based on this reasoning.

- 2) The declared type of a <graph reference value expression> shall be a graph reference value type.
- 3) The declared type of a <binding table reference value expression> shall be a binding table reference value type.
- 4) The declared type of a <node reference value expression> shall be a node reference value type.
- 5) The declared type of an <edge reference value expression> shall be an edge reference value type.
- 6) The declared type of a <record expression> shall be a record type.
- 7) If a <value expression> that is not immediately contained in an <aggregating value expression> directly contains an <aggregate function> *AF*, then:
 - a) The <value expression> or the <dependent value expression> immediately contained in *AF* shall contain one or more <binding variable reference>s to exactly one variable whose declared type is a material group list value type.
 - b) All <binding variable reference>s simply contained in *AF* whose declared type is a group list value type shall have the same name.

General Rules

None.

Conformance Rules

- 1) Without Feature GV45, “Record types”, conforming GQL language shall not contain a <record expression>.
- 2) Without Feature GV60, “Graph reference value types”, conforming GQL language shall not contain a <graph reference value expression>.
- 3) Without Feature GV61, “Binding table reference value types”, conforming GQL language shall not contain a <binding table reference value expression>.
- 4) Without Feature GE09, “Horizontal aggregation”, conforming GQL language shall not contain a <value expression> not immediately contained in an <aggregating value expression> that directly contains an <aggregate function>.

20.2 <value expression primary>

Function

Specify a value that is syntactically self-delimited.

Format

```
<value expression primary> ::=  
    <parenthesized value expression>  
    | <non-parenthesized value expression primary>  
  
<parenthesized value expression> ::=  
    <left paren> <value expression> <right paren>  
  
<non-parenthesized value expression primary> ::=  
    <non-parenthesized value expression primary special case>  
    | <binding variable reference>  
  
<non-parenthesized value expression primary special case> ::=  
    <aggregate function>  
    | <unsigned value specification>  
    | <list value constructor>  
    | <record constructor>  
    | <path value constructor>  
    | <property reference>  
    | <value query expression>  
    | <case expression>  
    | <cast specification>  
    | <element_id function>  
    | <let value expression>
```

Syntax Rules

None.

General Rules

None.

Conformance Rules

- 1) Without Feature GE01, “Graph reference value expressions”, conforming GQL language shall not contain a <value expression> that simply contains a <graph reference value expression>.
- 2) Without Feature GE02, “Binding table reference value expressions”, conforming GQL language shall not contain a <value expression> that simply contains a <binding table reference value expression>.

20.3 <value specification>

Function

Specify a value.

Format

```
<value specification> ::=  
    <literal>  
    | <general value specification>  
  
<unsigned value specification> ::=  
    <unsigned literal>  
    | <general value specification>  
  
<non-negative integer specification> ::=  
    <unsigned integer>  
    | <dynamic parameter specification>  
  
<general value specification> ::=  
    <dynamic parameter specification>  
    | SESSION_USER
```

Syntax Rules

- 1) The declared type of <non-negative integer specification> is an implementation-defined (ID062) exact numeric type.
- 2) The declared type of SESSION_USER is an implementation-defined (ID061) character string type.

General Rules

- 1) A <value specification>, <unsigned value specification>, or <non-negative integer specification> specifies a value that is not matched from a graph or selected from a binding table.
- 2) If a <non-negative integer specification> *NNIS* is specified, then:
 - a) Let *NNIV* be defined as follows.
Case:
 - i) If *NNIS* is an <unsigned integer> *UI*, then *NNIV* is the value of *UI*.
 - ii) Otherwise, *NNIS* is a <dynamic parameter specification> *DPS*:
 - 1) Let *PV* be the result of *DPS*.
 - 2) If *PV* is the null value, then an exception condition is raised: *data exception — null value not allowed (22004)*.
 - 3) If *PV* is not an exact number with scale 0 (zero), then an exception condition is raised: *data exception — invalid value type (22G03)*.
 - 4) If *PV* is negative, then an exception condition is raised: *data exception — negative limit value (22G02)*.
 - 5) *NNIV* is *PV*.

- b) If *NNIV* is not included in the declared type of *NNIS*, then an exception condition is raised:
data exception — invalid value type (22G03).
 - c) The result of *NNIS* is *NNIV*.
- 3) The value specified by `SESSION_USER` is the <authorization identifier> that identifies the session authorization identifier.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

20.4 <dynamic parameter specification>

Function

Specify a dynamic parameter for a value.

Format

<dynamic parameter specification> ::=
 <general parameter reference>

Syntax Rules

- 1) Let *DPS* be <dynamic parameter specification>.
- 2) Let *PN* and *PVT* be the parameter name and parameter value type, respectively, of *DPS*.
 NOTE 311 — The parameter name and the parameter value type of a <dynamic parameter specification> is determined by the Syntax Rules of [Subclause 22.1](#), “Annotation of a <GQL-program>”.
- 3) The declared type of *DPS* is *PVT*.

General Rules

- 1) The result of *DPS* is the parameter value of the current dynamic parameter whose parameter name is *PN*.

Conformance Rules

- 1) Without Feature GE04, “Graph parameters”, in conforming GQL language, the declared type of a <dynamic parameter specification> shall not be a supertype of a graph reference value type.
- 2) Without Feature GE05, “Binding table parameters”, in conforming GQL language, the declared type of a <dynamic parameter specification> shall not be a supertype of a binding table reference value type.

20.5 <let value expression>

Function

Evaluate a value expression using an amended working record.

Format

```
<let value expression> ::=  
  LET <let variable definition list> IN <value expression> END
```

Syntax Rules

- 1) Let LVE be the <let value expression> and let WRT be the declared type of the incoming working record of LVE .
- 2) Let $LVDL$ be the <let variable definition list>. Let N be the number of elements of $LVDL$. For i , 1 (one) $\leq i \leq N$, let LVD_i be the i -th element of $LVDL$.
- 3) For i , 1 (one) $\leq i \leq N$:
 - a) Let FN_i be the name specified by the <binding variable> contained in LVD_i without an intervening instance of <value expression>.
 - b) FN_i shall not identify a field type of WRT .
 - c) Let FE_i be the <value expression> simply contained in LVD_i .
- 4) For all pairs of i, j such that 1 (one) $\leq i < j \leq N$: FN_i shall not be equal to FN_j .
- 5) Let NRT be a record types whose field types are defined as follows. For i , 1 (one) $\leq i \leq N$, the field type whose name is FN_i and whose value type is the declared type of FE_i is included in NRT .
- 6) Let RHS be the <value expression> immediately contained in LVE .
- 7) The declared type of the incoming working record of RHS is WRT amended with NRT .
- 8) The declared type of LVE is the declared type of RHS .

General Rules

- 1) Let NR be a record whose fields are defined as follows. For i , 1 (one) $\leq i \leq N$, the field whose name is FN_i and whose value type is the result of FE_i is included in NR .
- 2) In a new child execution context amended with NR , let $RESULT$ be the result of RHS .
- 3) The result of LVE is $RESULT$.

Conformance Rules

- 1) Without Feature GE03, "Let-binding of variables in expressions", conforming GQL language shall not contain a <let value expression>.

20.6 <value query expression>

Function

Specify a scalar value derived from a <nested query specification>.

Format

<value query expression> ::=
VALUE <nested query specification>

Syntax Rules

- 1) Let *VQE* be the <value query expression>.
- 2) Let *NQS* be the <nested query specification> simply contained in *VQE*.
- 3) The declared type of *NQS* shall be a binding table type with a single column *SC*.
- 4) The *immediately emitting statement* of a <nested query specification> *NQXXX* is the last <statement> simply contained in *NQXXX*.
- 5) The *emitting result statement* of a <nested query specification> *NQSX* is defined as follows:
 - a) The immediately emitting statement of *NQSX* shall be a <linear query statement>.
 - b) Case:
 - i) If the immediately emitting statement of *NQSX* implicitly or explicitly simply contains a <primitive result statement> *PRSX*, then the emitting result statement of *NQSX* is *PRSX*.
 - ii) If the immediately emitting statement of *NQSX* is a <focused nested query specification> or an <ambient linear query statement> that immediately contains a <nested query specification> *NNQSX* and the emitting result statement of *NNQSX* is defined, then the emitting result statement of *NQSX* is the emitting result statement of *NNQSX*.
 - iii) Otherwise, the emitting result statement of *NQSX* is “omitted”.
- NOTE 312 — This is a recursive definition.
- 6) The emitting result statement of *NQS* shall not be “omitted”.
- 7) Let *PRS* be the emitting result statement of *NQS*.
- 8) *PRS* shall implicitly or explicitly simply contain the <return statement> *RS*.
NOTE 313 — The emitting statement of *NQS* can be a <select statement> as long as the syntax transformation specified in Subclause 14.12, “<select statement>” yields a <linear query statement> that fulfills the Syntax Rules of this Subclause if it is taken as the emitting statement of *NQS*.
- 9) Let *RI* be the only <return item> simply contained in *RS*.
NOTE 314 — To fulfill Syntax Rule 3), *RS* can only contain a single <return item>.
- 10) One of the following shall be true:
 - a) *PRS* shall contain an <order by and page statement> that simply contains a <limit clause> LIMIT 1.

- b) *PRS* shall not simply contain a <group by clause> and *RI* shall directly contain an <aggregate function>.
- 11) The declared type of the incoming working record of *NQS* is the declared type of the incoming working record of *VQE*.
- 12) The declared type of the incoming working table of *NQS* is the material unit binding table type.
- 13) The declared type of *VQE* is the column type of the column *SC*.

NOTE 315 — To fulfill [Syntax Rule 3](#)), the declared type of *NQS* is a binding table type with a single column *SC*.

General Rules

- 1) In a new child execution context: The General Rules of [Subclause 9.2](#), “<procedure body>” are applied; let *RNQS* be the result of the application of these rules.
- 2) Case:
 - a) If *RNQS* is an empty binding table, then the result of *VQE* is the null value.
 - b) Otherwise, *RNQS* is a binding table comprising a single record with a single field *F* and the result of *VQE* is the value of *F*.

Conformance Rules

- 1) Without Feature GQ18, “Scalar subqueries”, conforming GQL language shall not contain a <value query expression>.

20.7 <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 list> THEN <result>

<searched when clause> ::=
    WHEN <search condition> THEN <result>

<else clause> ::=
    ELSE <result>

<case operand> ::=
    <non-parenthesized value expression primary>
  | <element variable reference>

<when operand list> ::=
    <when operand> [ { <comma> <when operand> }... ]

<when operand> ::=
    <non-parenthesized value expression primary>
  | <comparison predicate part 2>
  | <>null predicate part 2>
  | <value type predicate part 2>
  | <normalized predicate part 2>
  | <directed predicate part 2>
  | <labeled predicate part 2>
  | <source predicate part 2>
  | <destination predicate part 2>

<result> ::=
    <result expression>
  | <>null literal>

<result expression> ::=
    <value expression>
```

Syntax Rules

- 1) If a <case expression> specifies a <case abbreviation>, then:
 - a) Let NV be the number of <value expression>s directly contained in the <case abbreviation> and let V_i , $1 \text{ (one)} \leq i \leq NV$, be the i -th such <value expression>.
 - b) $\text{NULLIF } (V_1, V_2)$ is equivalent to the following <case specification>:


```
CASE WHEN
  V1=V2 THEN
  NULL ELSE V1
END
```
 - c) $\text{COALESCE } (V_1, V_2)$ is equivalent to the following <case specification>:


```
CASE
  WHEN NOT V1 IS NULL THEN V1
  ELSE V2
END
```
 - d) For any number $n \geq 3$ of <value expression>s, $\text{COALESCE } (V_1, V_2, \dots, V_n)$ is equivalent to the following <case specification>:


```
CASE
  WHEN NOT V1 IS NULL THEN V1
  ELSE COALESCE (V2, ..., Vn)
END
```

NOTE 316 — This is a recursive definition.
- 2) If a <case specification> specifies a <simple case>, then let CO be the <case operand>.
 - a) If any <when operand> is <directed predicate part 2>, <labeled predicate part 2>, <source predicate part 2> or <destination predicate part 2>, then CO shall be <element variable reference> and every <when operand> shall be <directed predicate part 2>, <labeled predicate part 2>, <source predicate part 2> or <destination predicate part 2>; otherwise, CO shall not be <element variable reference> and no <when operand> shall be <directed predicate part 2>, <labeled predicate part 2>, <source predicate part 2> or <destination predicate part 2>.
 - b) Let N be the number of <simple when clause>s.
 - c) For i , $1 \text{ (one)} \leq i \leq N$:
 - i) Let WOL_i be the <when operand list> of the i -th <simple when clause>.
 - ii) Let R_i be the <result> of the i -th <simple when clause>.
 - iii) Let $M(i)$ be the number of <when operand>s simply contained in WOL_i .
 - iv) For j , $1 \text{ (one)} \leq j \leq M(i)$:
 - 1) let $WO_{i,j}$ be the j -th <when operand> simply contained in WOL_i .
 - 2) Case:
 - A) If $WO_{i,j}$ is a <non-parenthesized value expression primary>, then let $EW O_{i,j}$ be:

$$= WO_{i,j}$$

B) Otherwise, let EWO_{ij} be WO_{ij} .

- d) If <else clause> is specified, then let $CEEC$ be that <else clause>; otherwise, let $CEEC$ be the zero-length character string.
- e) The <simple case> is equivalent to a <searched case> in which the i -th <searched when clause> takes the form:

WHEN ($CO\ EWO_{i,1}$) OR
... OR
($CO\ EWO_{i,M(i)}$)
THEN R_i

- f) The <else clause> of the equivalent <searched case> takes the form:

$CEEC$

- g) The Conformance Rules of the Subclauses of [Clause 19, “Predicates”](#), are applied to the result of this syntactic transformation.

NOTE 317 — The specific Subclauses of [Clause 19, “Predicates”](#), are determined by the predicates that are created as a result of the syntactic transformation.

- 3) At least one <result> in a <case specification> shall specify a <result expression>.
- 4) If an <else clause> is not specified, then ELSE NULL is implicit.
- 5) The Syntax Rules of [Subclause 22.18, “General combination of value types”](#), are applied with the set of declared types of all <result expression>s in the <case specification> as $DTSET$; let DT be the $RESTYPE$ returned from the application of those Syntax Rules. The declared type of the <case specification> is DT .

General Rules

- 1) Case:
- a) If a <result> specifies NULL, then its value is the null value.
- b) Otherwise, a <result> specifies a <value expression> and its value is the result of that <value expression>.
- 2) Case:
- a) If the result of the <search condition> of some <searched when clause> in a <case specification> is True, then the result of the <case expression> is the result of the <result> of the first (left-most) <searched when clause> whose <search condition> evaluates to True, cast as the declared type of the <case specification>.
- b) Otherwise, no <search condition> in a <case specification> evaluates to True and the result of the <case expression> is the result of the <result> of the explicit or implicit <else clause>, cast as the declared type of the <case specification>.

Conformance Rules

None.

20.8 <cast specification>

Function

Specify a data conversion.

Format

```
<cast specification> ::=  
    CAST <left paren> <cast operand> AS <cast target> <right paren>  
  
<cast operand> ::=  
    <value expression>  
    | <null literal>  
  
<cast target> ::=  
    <value type>
```

Syntax Rules

- 1) Let *CS* be the <cast specification>.
- 2) Let *CO* be the <cast operand>.
- 3) Let *TD* be the data type identified by <value type>.
- 4) *TD* shall not specify an immaterial value type.
- 5) If *CO* specifies NULL, then *TD* shall be nullable, the declared type of *CS* is *TD* and no further Syntax Rules of this Subclause are applied; otherwise, *TD* shall not be an immaterial value type.
- 6) The declared type of *CS* is *TD*.
- 7) If *CO* is a <value expression>, then let *SD* be the declared type of the <value expression>.
- 8) If *CO* is a <value expression>, then the valid combinations of *TD* and *SD* in *CS* are given in Table 4, "Valid combinations of source and target and types". "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 invalid. The combination of *TD* and *SD* shall not be invalid according to this classification.

Table 4 — Valid combinations of source and target and types

<i>SD</i>	<i>EN</i>	<i>UN</i>	<i>AN</i>	<i>C</i>	<i>D</i>	<i>T</i>	<i>DT</i>	<i>DU</i>	<i>TD</i> <i>BO</i>	<i>B</i>	<i>L</i>	<i>R</i>	<i>P</i>	<i>DY</i>	<i>GR</i>	<i>NR</i>	<i>ER</i>	<i>TR</i>
EN	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	M	N	N	N	N
UN	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	M	N	N	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	M	N	N	N	N
C	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	M	N	N	N	N
D	N	N	N	Y	Y	N	Y	N	N	N	N	N	N	M	N	N	N	N
T	N	N	N	Y	N	Y	Y	N	N	N	N	N	N	M	N	N	N	N
DT	N	N	N	Y	Y	Y	Y	N	N	N	N	N	N	M	N	N	N	N
DU	N	N	N	Y	N	N	N	Y	N	N	N	N	N	M	N	N	N	N
BO	N	N	N	Y	N	N	N	N	Y	N	N	N	N	M	N	N	N	N
B	N	N	N	N	N	N	N	N	N	Y	N	N	N	M	N	N	N	N
L	N	N	N	N	N	N	N	N	N	N	M	N	Y	M	N	N	N	N
R	N	N	N	N	N	N	N	N	N	N	N	M	N	M	N	N	N	N
P	N	N	N	N	N	N	N	N	N	N	N	N	Y	M	N	N	N	N
DY	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
GR	N	N	N	N	N	N	N	N	N	N	N	N	N	M	Y	N	N	N
NR	N	N	N	N	N	N	N	N	N	N	N	N	Y	M	N	Y	N	N
ER	N	N	N	N	N	N	N	N	N	N	N	N	Y	M	N	N	Y	N
TR	N	N	N	N	N	N	N	N	N	N	N	N	Y	M	N	N	N	Y

Where:

EN = Signed Exact Numeric
 UN = Unsigned Exact Numeric
 AN = Approximate Numeric
 C = Character String
 D = Date
 T = Zoned Time & Local Time
 DT = Zoned Datetime & Local Datetime
 DU = Duration
 BO = Boolean
 B = Byte String
 L = List Value
 R = Record
 P = Path Value
 DY = Dynamic Union
 GR = Graph Reference Value
 NR = Node Reference Value
 ER = Edge Reference Value
 TR = Binding Table Reference Value

- 9) If *TD* is a dynamic union type, then *SD* shall be a subtype of *TD*.
- 10) If *SD* is a dynamic union type and *TD* is a static value type, then there shall be a component type *SCT* of *SD* such that

CAST (*VALUE* AS *TD*)

 where *VALUE* is a <value expression> whose declared type is *SCT*, shall be a valid <cast specification>.
- 11) If *SD* is a list value type, then:
 - a) If *SD* is a regular list value type, then *TD* shall be a regular list value type.
 - b) Let *ESD* be the list element type of *SD*.
 - c) Let *ETD* be the list element type of *TD*.
 - d) The <cast specification>:

CAST (*VALUE* AS *ETD*)

where *VALUE* is a <value expression> of declared type *ESD*, shall be a valid <cast specification>.

12) If *SD* is a closed record type and *TD* is a closed record type, then:

- a) Let *TFNS* be the set of field type names of *TD*.
- b) The set of field type names of *SD* shall include *TFNS*.
- c) For every name *FN* in *TFNS*:
 - i) Let *SFVT* be the value type of the field type with name *FN* in *SD*.
 - ii) Let *TFVT* be the value type of the field type with name *FN* in *TD*.
 - iii) The <cast specification>:

CAST (*VALUE* AS *TFVT*)

where *VALUE* is a <value expression> of declared type *SFVT*, shall be a valid <cast specification>.

General Rules

1) If *CO* is a <value expression> *VE*, then let *SV* be the result of *VE*.

2) Case:

- a) If *CO* is NULL, then the result of *CS* is the null value and no further General Rules of this Subclause are applied.
- b) If *SV* is the null value, then

Case:

- i) If *TD* is nullable, then the result of *CS* is the null value and no further General Rules of this Subclause are applied.
- ii) Otherwise, *TD* is material and the following exception condition is raised: *data exception — invalid value type (22G03)*.

3) Let *STZ* be the current time zone displacement.

4) Let *TV* be defined as follows.

Case:

- a) If *TD* is a dynamic union type, then *TV* is *SV*.
- b) If *SD* is a dynamic union type and *TD* is a static value type, then:
 - i) Let *MST* be the most specific static value type of *SV*.
NOTE 318 — *MST* is never a dynamic union type.
 - ii) If CAST (*VALUE* AS *TD*), where *VALUE* is a <value expression> of declared type *MST*, is not a valid <cast specification>, then an exception condition is raised: *data exception — invalid value type (22G03)*.
 - iii) For the remaining General Rules of this Subclause, *SD* is effectively replaced by *MST*.
- c) If *TD* is a reference value type, then:

Case:

- i) If TD is a closed reference value type and the GQL-object referenced by SV is not an object of the constraining GQL-object type of TD , then an exception condition is raised: *data exception — invalid value type (22G03)*.
- ii) Otherwise, TV is SV .
- d) If TD is a list value type and SD is a list value type, then:
 - i) Let NS be the cardinality of SV .
 - ii) For i , $1 \text{ (one)} \leq i \leq NS$, let VE_i be the i -th element of SV .
 - iii) Let $TLET$ be the list element type of TD .
 - iv) For i , $1 \text{ (one)} \leq i \leq NS$, the following <cast specification> is applied:
 CAST (VE_i AS $TLET$)
 yielding value TVE_i .
 - v) Let NT be the maximum cardinality of TD .
 - vi) Case:
 - 1) If NS is greater than NT , then an exception condition is raised: *data exception — list data, right truncation (22G0B)*.
 - 2) Otherwise, TV is the list value with elements TVE_i , $1 \text{ (one)} \leq i \leq NS$.
- e) If TD is a record type and SD is a record type, then
 Case:
 - i) If TD is a closed record type, then:
 - 1) If the set of field type names of TD is included in the set of field names of SV , then:
 - A) Let N be the number of field types in TD .
 - B) For i , $1 \text{ (one)} \leq i \leq N$, let FN_i be the i -th name in a permutation of the set of field type names of TD , let SFV_i be the value of the field with name FN_i of SV and let $TFVT_i$ be the value type of the field type with name FN_i of TD .
 - C) TV is the record resulting from the evaluation of:

$$\text{RECORD } \{ \text{CAST } (SFV_1 \text{ AS } TFVT_1), \\ \text{CAST } (SFV_2 \text{ AS } TFVT_2) \\ \dots \\ \text{CAST } (SFV_N \text{ AS } TFVT_N) \}$$
 - 2) Otherwise, an exception condition is raised: *data exception — record fields do not match (22G0U)*.
 - ii) Otherwise, TV is SV .
- f) If TD is a path value type and SD is a list value type, then
 Case:
 - i) If SV includes the null value or does not identify a path, then an exception condition is raised: *data exception — malformed path (22G0Z)*.
 - ii) Otherwise, TV is the path value whose path element list is SV .

g) If *TD* is a signed exact numeric type, then

Case:

i) If *SD* is a numeric type, then

Case:

- 1) If there is a representation of *SV* in the value 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 (IA005).
- 2) Otherwise, an exception condition is raised: *data exception — numeric value out of range* (22003).

ii) If *SD* is a character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.

Case:

- 1) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 21.2, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast* (22018).
- 2) Otherwise, let *LT* be that <signed numeric literal> such that *TV* is the result of:
CAST (*LT* AS *TD*)

h) If *TD* is an unsigned exact numeric type, then

Case:

i) If *SD* is a numeric type, then

Case:

- 1) If there is a representation of *SV* in the value type *TD* that does not lose any leading significant digits or the sign after rounding or truncating if necessary, then *TV* is that representation. The choice of whether to round or truncate is implementation-defined (IA005).
- 2) Otherwise, an exception condition is raised: *data exception — numeric value out of range* (22003).

ii) If *SD* is a character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.

Case:

- 1) If *SV* does not comprise an <unsigned numeric literal> as defined by the rules for <literal> in Subclause 21.2, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast* (22018).
- 2) Otherwise, let *LT* be that <unsigned numeric literal>. such that *TV* is the result of:
CAST (*LT* AS *TD*)

i) If *TD* is an approximate numeric type, then

Case:

i) If *SD* is a numeric type, then

Case:

- 1) If there is a representation of *SV* in the value 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 (IA005).
 - 2) Otherwise, an exception condition is raised: *data exception — numeric value out of range* (22003).
- ii) If *SD* is a character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.

Case:

- 1) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 21.2, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast* (22018).
 - 2) Otherwise, let *LT* be that <signed numeric literal> such that *TV* is the result of:
CAST (*LT* AS *TD*)
- j) If *TD* is a character string type, then let *MINL* and *MAXL* be the minimum and maximum length, respectively, in characters of *TD*.

Case:

- i) If *SD* is an exact numeric type, then:
- 1) Let *YP* be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 21.2, "<literal>" that does not simply contain an <exact number suffix>, whose scale is the same as the scale of *SD*, whose interpreted value is the absolute value of *SV*, and that is not an <unsigned hexadecimal integer>.
 - 2) Case:
 - A) If *SV* is less than 0 (zero), then let *Y* be the result of '−' || *YP*.
 - B) Otherwise, let *Y* be *YP*.
 - 3) Case:
 - A) If the length in characters of *Y* is greater than *MAXL*, then an exception condition is raised: *data exception — string data, right truncation* (22001).
 - B) If the length in characters *M* of *Y* is less than *MINL*, then *TV* is *Y* extended on the right by *MINL*−*M* <space> characters.
 - C) Otherwise, the length in characters of *Y* is both greater than or equal to *MINL* and less than or equal to *MAXL*. *TV* is *Y*.
- ii) If *SD* is an approximate numeric type, then:
- 1) Let *YP* be a character string defined as follows.
- Case:
- A) If *SV* equals 0 (zero), then *YP* is '0E0'.
 - B) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 21.2, "<literal>" that does not simply contain an <approximate number suffix>, 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>.

2) Case:

A) If *SV* is less than 0 (zero), then let *Y* be the result of '−' || *YP*.

B) Otherwise, let *Y* be *YP*.

3) Case:

A) If the length in characters of *Y* is greater than *MAXL*, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

B) If the length in characters *M* of *Y* is less than *MINL*, then *TV* is *Y* extended on the right by *MINL*−*M* <space> characters.

C) Otherwise, the length in characters of *Y* is greater than or equal to *MINL* and less than or equal to *MAXL*. *TV* is *Y*.

iii) If *SD* is a character string type, then

Case:

1) If the length in characters of *SV* is greater than *MAXL*, then *TV* is the first *MAXL* characters of *SV*. If any of the remaining characters of *SV* are not <truncating whitespace> characters, then a completion condition is raised: *warning — string data, right truncation (01004)*.

2) If the length in characters *M* of *SV* is less than *MINL*, then *TV* is *SV* extended on the right by *MINL*−*M* <space> characters.

3) Otherwise, the length in characters of *SV* is both greater than or equal to *MINL* and less than or equal to *MAXL*. *TV* is *SV*.

iv) If *SD* is a temporal instant type or a temporal duration type, then:

1) Let *Y* be the shortest character string that conforms to the definition of <literal> in Subclause 21.2, "<literal>", and such that the interpreted value of *Y* is *SV* and the interpreted precision of *Y* is the precision of *SD*, and such that

Case:

A) If *SD* is a date type, then the character string includes the time scale components: [year], [month], and [day].

B) If *SD* is a local time type, then the character string includes the time scale components: [hour], [min], and [sec], but does not include the 'T' time designator.

C) If *SD* is a zoned time type, then the character string includes the time scale components: [hour], [min], [sec], and [shift], but does not include the 'T' time designator.

D) If *SD* is a local datetime type, then the character string includes the time scale components: [year], [month], [day], [hour], [min], and [sec].

E) If *SD* is a zoned datetime type, then the character string includes the time scale components: [year], [month], [day], [hour], [min], [sec], and [shift].

- F) If *SD* is a year and month-based duration, then the character string includes the time scale components: [year] and [month].
- G) If *SD* is a day and time-based duration, then the character string includes the time scale components: [day], [hour], [min], and [sec].
- 2) Case:
- A) If the length in characters of *Y* is greater than *MAXL*, then an exception condition is raised: *data exception — string data, right truncation (22001)*.
- B) If the length in characters *M* of *Y* is less than *MINL*, then *TV* is *Y* extended on the right by *MINL*−*M* <space> characters.
- C) Otherwise, the length in characters of *Y* is both greater than or equal to *MINL* and less than or equal to *MAXL*. *TV* is *SV*.
- v) If *SD* is a Boolean type, then:
- 1) Let *Y* be defined as follows. If *SV* is True, then *Y* is 'TRUE'; otherwise, *SV* is False and *Y* is 'FALSE'.
- 2) Case:
- A) If the length in characters of *Y* is greater than *MAXL*, then an exception condition is raised: *data exception — string data, right truncation (22001)*.
- B) If the length in characters *M* of *Y* is less than *MINL*, then *TV* is *Y* extended on the right by *MINL*−*M* <space> characters.
- C) Otherwise, the length in characters of *Y* is both greater than or equal to *MINL* and less than or equal to *MAXL*. *TV* is *SV*.
- k) If *TD* is the date type, then
- Case:
- i) If *SD* is character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.
- Case:
- 1) If the rules for <literal> in Subclause 21.2, "<literal>", can be applied to *SV* to determine a valid value of the value type *TD*, then let *TV* be that value.
- 2) Otherwise, an exception condition is raised: *data exception — invalid date, time, or, datetime format (22007)*.
- ii) If *SD* is the date type, then *TV* is *SV*.
- iii) If *SD* is a datetime type, then let *DS* be the character string obtained by extracting the first 8 characters from the result of:
CAST (*VE* AS STRING)
TV is the result of:
DATE (*DS*)
- l) If *TD* is a local time type, then
- Case:
- i) If *SD* is a character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.

Case:

- 1) If the rules for <literal> in Subclause 21.2, "<literal>", can be applied to *SV* to determine a valid value of the value type *TD*, then let *TV* be that value.
 - 2) Otherwise, an exception condition is raised: *data exception — invalid date, time, or, datetime format (22007)*.
- ii) If *SD* is a local time type, then *TV* is *SV*.
- iii) If *SD* is a zoned time type, then:
- 1) Let *TIME* be the result of:
CAST (*VE* AS STRING)
 - 2) If the last character of *TIME* is 'Z', then the last character of *TIME* is effectively replaced by '+0000'.
 - 3) Let *TZ* be the character string obtained by extracting the last 5 characters from *TIME*.
 - 4) Let *H* be the character string obtained by extracting the first 3 characters from *TZ*.
 - 5) Let *M* be the character string obtained by extracting the 4th and 5th characters from *TZ*.
 - 6) Let *TZD* be:
DURATION ('P' || *H* || 'H' || *M* || 'M')
 - 7) Let *TL* be the result of CHARACTER_LENGTH (*TIME*).
 - 8) Let *T* be the character string obtained by extracting the first *TL* – 5 characters from *TIME*.
 - 9) *TV* is the result of:
LOCAL_TIME (*T*) + *TZD*
.
- iv) If *SD* is a local datetime type, then:
- 1) Let *T* be the character string obtained by extracting the first 10 characters from the result of:
CAST (*VE* AS STRING)
 - 2) *TV* is the result of:
LOCAL_TIME (*T*)
- v) If *SD* is a zoned datetime type, then:
- 1) Let *TIME* be the character string obtained by extracting the first 10 characters from the result of:
CAST (*VE* AS STRING)
If the last character of *TIME* is 'Z', then the last character of *TIME* is effectively replaced by '+0000'.
 - 2) Let *TZ* be the character string obtained by extracting the last 5 characters from *TIME*.
 - 3) Let *H* be the character string obtained by extracting the first 3 characters from *TZ*.

- 4) Let M be the character string obtained by extracting the 4th and 5th characters from TZ .
 - 5) Let TZD be:
DURATION ('P' || H || 'H' || M || 'M')
 - 6) Let TL be the result of:
CHARACTER_LENGTH ($TIME$)
 - 7) Let T be the character string obtained by extracting the first $TL - 5$ characters from $TIME$.
 - 8) TV is the result of LOCAL_TIME (T) + TZD .
- m) If TD is a zoned time type, then
- Case:
- i) If SD is a character string type, then SV is effectively replaced by SV with any leading or trailing <truncating whitespace> removed.
Case:
 - 1) If the rules for <literal> in Subclause 21.2, "<literal>", can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
 - 2) Otherwise, an exception condition is raised: *data exception — invalid date, time, or, datetime format (22007)*.
 - ii) If SD is a zoned time type, then TV is SV .
 - iii) If SD is a local time type, then:
 - 1) Let T be the result of:
CAST (VE AS STRING)
 - 2) TV is the result of:
ZONED_TIME (T || STZ)
 - iv) If SD is a zoned datetime type, then:
 - 1) Let ZDT be the result of:
CAST (VE AS STRING)
 - 2) Let $LZDT$ be the result of:
CHARACTER_LENGTH (ZDT)
 - 3) Let T be the character string obtained by extracting the last $LZDT - 10$ characters from ZDT .
 - 4) TV is the result of:
ZONED_TIME (T)
 - v) If SD is a local datetime type, then:
 - 1) Let ZDT be the result of:
CAST (VE AS STRING)
 - 2) Let $LZDT$ be the result of:
CHARACTER_LENGTH (ZDT)

- 3) T be the character string obtained by extracting the last $LZDT - 10$ characters from ZDT .
 - 4) TV is the result of $ZONED\_TIME (T || STZ)$.
- n) If TD is a local datetime type, then
- Case:
- i) If SD is a character string type, then SV is effectively replaced by SV with any leading or trailing <truncating whitespace> removed.
- Case:
- 1) If the rules for <literal> in Subclause 21.2, "<literal>", can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
 - 2) Otherwise, an exception condition is raised: *data exception — invalid date, time, or, datetime format (22007)*.
- ii) If SD is a date type, then let D be the result of:
 $CAST (VE AS STRING)$
 TV is the result of:
 $LOCAL\_DATETIME (D || 'T000000')$
- iii) If SD is a local time type, then:
- 1) Let CD be the result of:
 $CAST (DATE() AS STRING)$
 - 2) Let T be the result of:
 $CAST (VE AS STRING)$
 - 3) TV is the result of:
 $LOCAL\_DATETIME (CD || 'T' || T)$
- iv) If SD is a zoned time type, then:
- 1) Let $DATETIME$ be the result of:
 $CAST (VE AS STRING)$
 If the last character of $DATETIME$ is 'Z', then the last character of $DATETIME$ is effectively replaced by '+0000'.
 - 2) Let TZ be the character string obtained by extracting the last 5 characters from $DATETIME$.
 - 3) Let H be the character string obtained by extracting the first 3 characters from TZ .
 - 4) Let M be the character string obtained by extracting the 4th and 5th characters from TZ .
 - 5) Let TZD be:
 $DURATION ('P' || H || 'H' || M || 'M')$
 - 6) Let T be the character string obtained by extracting the first 6 characters from $DATETIME$.
 - 7) Let CD be the result of:
 $CAST (DATE() AS STRING)$

ISO/IEC 39075:2024(en)
20.8 <cast specification>

- 8) Let *TA* be the result of:
CAST (LOCAL_TIME (*T*) + *TZD*) AS STRING)
- 9) *TV* is the result of LOCAL_DATETIME (*CD* || 'T' || *TA*).
- v) If *SD* is a local datetime type, then *TV* is *SV*.
- vi) If *SD* is a zoned datetime type, then:
- 1) Let *ZDT* be the result of:
CAST (*VE* AS STRING)
If the last character of *ZDT* is 'Z', then the last character of *TIME* is effectively replaced by '+0000'.
 - 2) Let *TZ* be the character string obtained by extracting the last 5 characters from *ZDT*.
 - 3) Let *H* be the character string obtained by extracting the first 3 characters from *TZ*.
 - 4) Let *M* be the character string obtained by extracting the 4th and 5th characters from *TZ*.
 - 5) Let *TZD* be:
DURATION ('P' || *H* || 'H' || *M* || 'M')
 - 6) Let *TL* be the result of:
CHARACTER_LENGTH (*TIME*)
 - 7) Let *DT* be the character string obtained by extracting the first 15 characters from the result of:
CAST (*VE* AS STRING)
 - 8) *TV* is the result of:
LOCAL_DATETIME (*DT*) + *TZD*
- o) If *TD* is a zoned datetime type, then
- Case:
- i) If *SD* is a character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.
- Case:
- 1) If the rules for <literal> in [Subclause 21.2](#), "<literal>", can be applied to *SV* to determine a valid value of the value type *TD*, then let *TV* be that value.
 - 2) Otherwise, an exception condition is raised: *data exception — invalid date, time, or, datetime format (22007)*.
- ii) If *SD* is a date type, then let *D* be the result of:
CAST (*VE* AS STRING)
TV is the result of:
ZONED_DATETIME (*D* || 'T000000' || *STZ*)
- iii) If *SD* is a local time type, then:
- 1) Let *CD* be the result of:
CAST (DATE() AS STRING)

- 2) Let *T* be the result of:
CAST (*VE* AS STRING)
- 3) *TV* is the result of
ZONED_DATETIME (*CD* || 'T' || *T* || *STZ*)
- iv) If *SD* is a zoned time type, then:
 - 1) Let *CD* be the result of:
CAST (DATE() AS STRING)
 - 2) Let *T* be the result of:
CAST (*VE* AS STRING)
 - 3) *TV* is the result of:
ZONED_DATETIME (*CD* || 'T' || *T*)
- v) If *SD* is a local datetime type, then:
 - 1) Let *DT* be the result of:
CAST (*VE* AS STRING)
 - 2) *TV* is the result of:
ZONED_DATETIME (*DT* || *STZ*)
- vi) If *SD* is a zoned datetime type, then *TV* is *SV*.
- p) If *TD* is a temporal duration type, then
Case:
 - i) If *SD* is a character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.
Case:
 - 1) If the rules for <literal> in Subclause 21.2, “<literal>”, can be applied to *SV* to determine a valid value of the value type *TD*, then let *TV* be that value.
 - 2) Otherwise, an exception condition is raised: *data exception — invalid duration format (22G0H)*.
 - ii) If *SD* is a temporal duration type, then *TV* is *SV*.
- q) If *TD* is a Boolean type, then
Case:
 - i) If *SD* is a character string type, then *SV* is effectively replaced by *SV* with any leading or trailing <truncating whitespace> removed.
Case:
 - 1) If the rules for <literal> in Subclause 21.2, “<literal>”, can be applied to *SV* to determine a valid value of the value type *TD*, then let *TV* be that value.
 - 2) Otherwise, an exception condition is raised: *data exception — invalid character value for cast (22018)*.
 - ii) If *SD* is a Boolean type, then *TV* is *SV*.
- r) If *TD* and *SD* are byte string types, then:

- i) Let *MINLTD* and *MAXLTD* be the minimum length and the maximum length, respectively, in bytes of *TD*.
 - ii) Case:
 - 1) If the length in bytes of *SV* is greater than *MAXLTD*, then *TV* is the first *MAXLTD* bytes of *SV* and a completion condition is raised: *warning — string data, right truncation (01004)*.
 - 2) If the length in bytes *M* of *SV* is smaller than *MINLTD*, then *TV* is *SV* extended on the right by *MINLTD – M* '00's.
 - 3) Otherwise, the length in bytes of *SV* is greater than or equal to *MINLTD* as well as less than or equal to *MAXLTD* and *TV* is *SV*.
- 5) The result of *CS* is *TV*.

Conformance Rules

- 1) Without Feature GA05, "Cast specification", conforming GQL language shall not contain a <cast specification>.

20.9 <aggregate function>

Function

Specify a value computed from a collection of records.

Format

```
<aggregate function> ::=
    COUNT <left paren> <asterisk> <right paren>
    | <general set function>
    | <binary set function>

<general set function> ::=
    <general set function type>
    <left paren> [ <set quantifier> ] <value expression> <right paren>

<binary set function> ::=
    <binary set function type>
    <left paren> <dependent value expression> <comma> <independent value expression>
    <right paren>

<general set function type> ::=
    AVG
    | COUNT
    | MAX
    | MIN
    | SUM
    | COLLECT_LIST
    | STDDEV_SAMP
    | STDDEV_POP

<set quantifier> ::=
    DISTINCT
    | ALL

<binary set function type> ::=
    PERCENTILE_CONT
    | PERCENTILE_DISC

<dependent value expression> ::=
    [ <set quantifier> ] <numeric value expression>

<independent value expression> ::=
    <numeric value expression>
```

Syntax Rules

- 1) Let *AF* be the <aggregate function>.
- 2) If *AF* immediately contains a <general set function> that does not specify the <set quantifier>, then ALL is implicit.
- 3) *AF* shall not contain a <procedure body>.
- 4) *AF* shall not simply contain an <aggregate function>.
- 5) Let the <value expression> or the <dependent value expression> *VE* be defined as follows.
Case:

ISO/IEC 39075:2024(en)
20.9 <aggregate function>

- a) If *AF* immediately contains a <general set function> *GSF*, then *VE* is the <value expression> immediately contained in *GSF*.
 - b) If *AF* immediately contains a <binary set function> *BSF*, then *VE* is the <dependent value expression> immediately contained in *BSF*.
 - c) Otherwise, *VE* is the <value expression> or the <dependent value expression> immediately contained in *AF*.
- 6) Let *DT* be the declared type of *VE*.
- 7) If *AF* immediately contains COUNT, then:
- a) If *AF* immediately contains <asterisk>, then *AF* shall be directly contained in an <aggregating value expression>.
 - b) The declared type of the result is an implementation-defined (ID059) exact numeric type with scale 0 (zero).
- 8) If *AF* immediately contains a <general set function>, then:
- a) If *AF* specifies a <general set function> whose <set quantifier> is DISTINCT, then *VE* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 22.15, “Grouping operations”, apply.
 - b) If *AF* specifies a <general set function type> that is MAX or MIN, then *VE* is an operand of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 22.14, “Ordering operations”, apply.
 - c) If MAX or MIN is specified, then the declared type of the result is *DT*.
 - d) If SUM or AVG is specified, then:
 - i) *DT* shall be a numeric type.
 - ii) If SUM is specified and *DT* is exact numeric with scale *S*, then the declared type of the result is an implementation-defined (ID095) exact numeric type with scale *S*.
 - iii) If AVG is specified and *DT* is exact numeric, then the declared type of the result is an implementation-defined (ID096) exact numeric type with precision not less than the precision of *DT* and scale not less than the scale of *DT*.
 - iv) If *DT* is approximate numeric, then the declared type of the result is an implementation-defined (ID097) approximate numeric type with precision not less than the precision of *DT*.
 - e) If STDDEV_POP or STDDEV_SAMP is specified, then *DT* shall be a numeric type and the declared type of the result shall be an implementation-defined (ID098) approximate numeric type. If *DT* is an approximate numeric type, then the precision of the result is not less than the precision of *DT*.
 - f) If COLLECT_LIST is specified, then the declared type of the result is the regular list value type whose list element type is *DT*.
- 9) If *AF* immediately contains a <binary set function>, then:
- a) Let *DVEXP* be the <numeric value expression> immediately contained in the <dependent value expression> immediately contained in *AF*.
 - b) *DVEXP* is an operand of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 22.14, “Ordering operations”, apply.

- c) Let *IVE* be the <independent value expression> simply contained in *AF*.
- d) If *AF* specifies no <set quantifier>, then *ALL* is implicit.
- e) The declared type of the incoming working record of the *DVEXP* is the declared type of the incoming working record of *AF* amended with the record type of declared type of the incoming working table of *AF*.
- f) The declared type of the incoming working table of *DVEXP* is the material unit binding table type.
- g) The declared type of the incoming working record of *IVE* is the declared type of the incoming working record of *AF*.
- h) The declared type of the incoming working table of *IVE* is the material unit binding table type.
- i) Let *DTIVE* be the declared type of *IVE*.
- j) The declared type of the result is an implementation-defined (ID099) approximate numeric type. If *DT* is an approximate numeric type, then the precision of the result is not less than the precision of *DT*. If *DTIVE* is an approximate numeric type, then the precision of the result is not less than the precision of *DTIVE*.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) If *AF* is *COUNT(*)*, then the result of *AF* is the count of records in *TABLE* and no further General Rules are applied.
- 3) If, during the computation of the result of *AF*, an intermediate result is not representable in the declared type of the site that contains that intermediate result, then
Case:
 - a) If the most specific static value type of the result of *AF* is a list value type, then an exception condition is raised: *data exception — list data, right truncation (22G0B)*.
 - b) If the most specific static value type of the result of *AF* is either a character string type or a byte string type, then an exception condition is raised: *data exception — string data, right truncation (22001)*.
 - c) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.
- 4) Let *SQ* be defined as follows.
Case:
 - a) If *AF* is a <general set function> *GSF*, then *SQ* is the <set quantifier> immediately contained in *GSF*.
 - b) Otherwise, *AF* is a <binary set function> *BSF* and *SQ* is the <set quantifier> immediately contained in the <dependent value expression> immediately contained in *BSF*.
- 5) Let the collection *VALUES* be determined as follows.
Case:
 - a) If *AF* is directly contained in a <value expression> that is not immediately contained in an <aggregating value expression>, then:

- i) Let *GLBV* be the binding variable referenced by <binding variable reference>s simply contained in *AF* without an intervening instance of <independent value expression> whose declared type is the group list value type.
 - ii) General Rules of Subclause 22.7, “Evaluation of an expression on a group variable”, are applied with *GLBV* as *GROUP LIST BINDING VARIABLE* and *VE* as *EXPRESSION*; let *VALUES* be the *LIST VALUE* returned from the application of those General Rules.
- b) Otherwise:
- i) Initially, *VALUES* is an empty collection.
 - ii) For each record *R* of *TABLE* in a new child execution context amended with *R*:
 - 1) Let *EXPRES* be the result of *VE*.
 - 2) Case:
 - A) If *EXPRES* is null, then a completion condition is raised: *warning — null value eliminated in set function (01G11)*.
 - B) Otherwise,
 - Case:
 - I) If *SQ* is *DISTINCT* and *EXPRES* is not in *VALUES*, then *EXPRES* is added to *VALUES*;
 - II) Otherwise, *SQ* is *ALL* and *EXPRES* is added to *VALUES*.
- 6) Let *N* be the cardinality of *VALUES*.
- 7) Let *RESULT* be defined as follows.
- Case:
- a) If *AF* is the <general set function> *GSF*, then
 - Case:
 - i) If *COUNT* is specified, then *RESULT* is *N*.
 - ii) If *VALUES* is empty, then *RESULT* is defined as follows.
 - Case:
 - 1) If *AF* is *COLLECT_LIST*, then *RESULT* is the empty list.
 - 2) Otherwise, *RESULT* is the null value.
 - iii) If *MAX* or *MIN* is specified, then *RESULT* is the result, respectively, of the maximum value or the minimum value in *VALUES*. *RESULT* is determined using the comparison rules specified in Subclause 19.3, “<comparison predicate>”.
 - iv) If *SUM* is specified, then *RESULT* is the sum of the values in *VALUES*. If *RESULT* is not within the range of the declared type of *RESULT*, then an exception condition is raised: *data exception — numeric value out of range (22003)*.
 - v) If *STDDEV_POP* or *STDDEV_SAMP* is specified, then let *SX* be the sum of values in *VALUES* and let *SXS* be the sum of the squares of the values in *VALUES*.
 - 1) If *STDDEV_POP* is specified, then *RESULT* is the result of $\text{SQRT}((\text{SXS} - \text{SX} * \text{SX} / \text{N}) / \text{N})$.
 - 2) If *STDDEV_SAMP* is specified, then

Case:

- A) If N is 1 (one), then *RESULT* is the null value.
 - B) Otherwise, then *RESULT* is the result of $\text{SQRT}((SXS - SX * SX / N) / (N - 1))$.
- 3) If *COLLECT_LIST* is specified, then *RESULT* is the list comprised of all values in *VALUES*. If *RESULT* is not within the range of the declared type of *RESULT*, then an exception condition is raised: *data exception — list data, right truncation (22G0B)*.

- b) Otherwise, *AF* is the <binary set function> *BSF*. Let *IVE* be the <independent value expression> immediately contained in *BSF*, let *IVERE* be the result of evaluating *IVE* in a new child execution context, and

Case:

- i) If *VALUES* is empty, then *RESULT* is the null value.
- ii) Let *ORDERED_VALUES* be the sequence of non-null elements in *VALUES* ordered from least to greatest. *ORDERED_VALUES* is determined using the comparison rules specified in Subclause 19.3, "<comparison predicate>". Let $N\_ORDERED\_VALUES$ be the cardinality of *ORDERED_VALUES*.
- iii) Let *INDEX* be the result of $1 + (IVERE * (N\_ORDERED\_VALUES - 1))$.
- iv) If *PERCENTILE_CONT* is specified, then

Case:

- 1) If *INDEX* is an integer, then *RESULT* is the value in *ORDERED_VALUES* at position *INDEX*.
- 2) Otherwise:
 - A) Let *INDEX_FLOOR* be the largest integer less than *INDEX* and let *RESULT_FLOOR* be the value in *ORDERED_VALUES* at position *INDEX_FLOOR*.
 - B) Let *INDEX_CEILING* be the smallest integer greater than *INDEX* and let *RESULT_CEILING* be the value in *ORDERED_VALUES* at position *INDEX_CEILING*.
 - C) *RESULT* is the result of $(INDEX\_CEILING - INDEX) * RESULT\_FLOOR + (INDEX - INDEX\_FLOOR) * RESULT\_CEILING$.

- v) If *PERCENTILE_DISC* is specified, then

Case:

- 1) If *INDEX* is an integer, then *RESULT* is the value in *ORDERED_VALUES* at position *INDEX*.
- 2) Otherwise, *RESULT* is the value in *ORDERED_VALUES* at the position obtained as a result of rounding or truncating *INDEX*. The choice of whether to round or truncate is implementation-defined (IA005).

- 8) The result of evaluating *AF* is *RESULT*.

Conformance Rules

- 1) Without Feature GF10, “Advanced aggregate functions: general set functions”, conforming GQL language shall not contain an <aggregate function> that immediately contains a <general set function type> that is that is COLLECT_LIST, STDDEV_SAMP, or STDDEV_POP.
- 2) Without Feature GF11, “Advanced aggregate functions: binary set functions”, conforming GQL language shall not contain an <aggregate function> that immediately contains a <binary set function type>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

20.10 <element_id function>

Function

Generate a unique identifier for a graph element.

Format

```
<element_id function> ::=  
  ELEMENT_ID <left paren> <element variable reference> <right paren>
```

Syntax Rules

- 1) Let *EIF* be the <element_id function>.
- 2) Let *EVR* be the <element variable reference> simply contained in *EIF*. *EIF* shall have singleton degree of reference.
- 3) The declared type of <element_id function> is an implementation-defined (ID076) type that is permitted as the declared type of an operand of an equality operation according to the Syntax Rules of Subclause 22.13, “Equality operations”.

General Rules

- 1) Let *GRV* be the result of *EVR*.
- 2) The result of *EIF* is defined as follows.

Case:

- a) If *GRV* is the null value, then the result of *EIF* is the null value.
- b) Otherwise, the result of *EIF* is an implementation-dependent (UV004) value that encapsulates the identity of the referent of *GRV* for the duration of the currently executing GQL-request.

NOTE 319 — The result of *EIF* can be but is not guaranteed to be the global object identifier of the referent of *GRV*.

Conformance Rules

- 1) Without Feature G100, “ELEMENT_ID function”, conforming GQL language shall not contain an <element_id function>.

20.11 <property reference>

Function

Reference a property of a graph element or a field of a record.

Format

```
<property reference> ::=  
  <property source> <period> <property name>  
  
<property source> ::=  
  <node reference value expression>  
  | <edge reference value expression>  
  | <record expression>
```

Syntax Rules

- 1) Let *PR* be the <property reference>, let *PS* be the <property source> immediately contained in *PR*, let *PSD* be the declared type of *PS*, and let *PN* be the <property name>, immediately contained in *PR*.
- 2) If *PS* is a <record expression>, then
Case:
 - a) If *PSD* is a closed record type, then *PN* shall identify a field type *FIELD* in *PSD* and the declared type of *PR* is the value type of *FIELD*.
 - b) Otherwise, *PSD* is an open record type and the declared type of *PR* is the open dynamic union type.
- 3) If *PS* is a <node reference value expression> or an <edge reference value expression>, then
Case:
 - a) If *PSD* is a closed graph element reference value type, then *PN* shall identify a property type *PT* in the constraining GQL-object type of *PSD* and the declared type of *PR* is the value type of *PT*.
 - b) Otherwise, *PSD* is an open graph element reference value type and the declared type of *PR* is the dynamic property value type.

General Rules

- 1) Let *SOURCE* be the result of *PS*.
- 2) Let *PRD* be the declared type of *PR*.
- 3) Case:
 - a) If *SOURCE* is the null value, then
Case:
 - i) If *PRD* is nullable, then the result of *PR* is the null value and no further General Rules of this Subclause are applied.

- ii) Otherwise, the following exception is raised: *data exception — invalid value type (22G12)*.
- b) If *SOURCE* is a record and *PN* identifies a field *FIELD* in *SOURCE*, then the result of *PR* is the value of *FIELD*.
- c) If *SOURCE* is a reference to a graph element *GE* and *PN* identifies a property *PROP* in *GE*, then:
 - i) Let *PV* be the property value of *PROP*.
 - ii) If *PV* is not a value of *PRD*, then an exception condition is raised: *data exception — invalid value type (22G12)*; otherwise, the result of *PR* is *PV*.
- d) Otherwise, either *SOURCE* is a record and *PN* does not identify a field in *SOURCE* or *SOURCE* is a graph element *GE* and *PN* does not identify a property in *GE*.

Case:

 - i) If *PRD* is nullable, then the result of *PR* is the null value.
 - ii) Otherwise, *PRD* is material and an exception condition is raised: *data exception — invalid value type (22G12)*.

Conformance Rules

- 1) Without Feature GV45, “Record types”, conforming GQL language shall not contain a <record expression>.

20.12 <binding variable reference>

Function

Specify binding variable references.

Format

<binding variable reference> ::=
 <binding variable>

Syntax Rules

- 1) Let *BVR* be the <binding variable reference>.
- 2) *BVR* specifies a *binding variable reference*.
- 3) Let *BV* be the <binding variable> immediately contained in *BVR*.
- 4) The *referenced binding variable* of *BVR* is the binding variable identified by *BVR*.

NOTE 320 — See Subclause 21.1, “Names and variables”.

- 5) The name of *BVR* is the name of the referenced binding variable of *BVR*.
- 6) Let *BVN* be the name of *BVR*.
- 7) Let *IREDT* be the declared type of the incoming working record of *BVR*.
- 8) *IREDT* shall have a field type whose name is *BVN*. Let *BVFT* be that field type.

NOTE 321 — The field value type of the incoming working record corresponding to a referenced binding variable *ABV* of a <binding variable reference> that is declared by a <graph pattern> *AGP* is defined by Syntax Rule 25) of Subclause 16.3, “<graph pattern binding table>” to be the same as the projected field type of the <binding variable reference> to *ABV* in the <graph pattern yield clause> of *AGP*. See Syntax Rule 13), for the definition of projected field type.

- 9) The declared type of *BVR* is the value type of *BVFT*.
- 10) The *binding graph pattern* of *BVR* is defined, if existing, as the innermost <graph pattern> *GP* for which it holds that:
 - a) *GP* declares an <element variable>, a <path variable>, or a <subpath variable> equivalent to *BV*.
 - b) At least one of the following is true:
 - i) *BVR* is simply contained in the <graph pattern where clause> of *GP*.
 - ii) *BVR* is contained in a <parenthesized path pattern where clause> simply contained in *GP*.
 - iii) *BVR* is contained in a <graph pattern yield clause> that is simply contained in a <graph pattern binding table> that simply contains *GP*.

NOTE 322 — This Syntax Rule is applied after the syntactic transform that converts any <element pattern where clause> to a <parenthesized path pattern where clause>.

- 11) The *degree of reference* of *BVR* is defined as follows.

Case:

ISO/IEC 39075:2024(en)
20.12 <binding variable reference>

- a) If the binding graph pattern *GP* of *BVR* is defined and declares an <element variable> *EV* equivalent to *BV*, then

Case:

- i) If *BVR* is simply contained in the <graph pattern where clause> of *GP*, then the degree of reference of *BVR* is the degree of exposure of *EV* by *GP*.

NOTE 323 — “Degree of exposure” is defined in Subclause 16.4, “<graph pattern>” and Subclause 16.7, “<path pattern expression>”.

- ii) If *BVR* is contained in a <parenthesized path pattern where clause> *PPPWC* simply contained in *GP*, then let *PPPE* be the <parenthesized path pattern expression> that simply contains *PPPWC*.

NOTE 324 — This rule is applied after the syntactic transform that converts any <element pattern where clause> to a <parenthesized path pattern where clause>.

Case:

- 1) If *EV* is declared by *PPPE*, then the degree of reference of *BVR* is the degree of exposure of *EV* by *PPPE*.

NOTE 325 — “Degree of exposure” is defined in Subclause 16.4, “<graph pattern>” and Subclause 16.7, “<path pattern expression>”.

- 2) Otherwise, let *PP* be the innermost <graph pattern> or <parenthesized path pattern expression> that contains *PPPWC* and that declares *EV*. The degree of reference of *BVR* is the degree of exposure of *EV* by *PP*. The degree of reference of *BVR* shall be singleton.

NOTE 326 — “Degree of exposure” is defined in Subclause 16.4, “<graph pattern>” and Subclause 16.7, “<path pattern expression>”.

- b) If the binding graph pattern *GP* of *BVR* is defined and declares a <path variable> equivalent to *BV*, then the degree of reference of *BVR* is unconditional singleton.

- c) Otherwise, the degree of reference of *BVR* is defined as follows.

Case:

- i) If the declared type of *BVR* is a group list value type or a dynamic union type whose component types are group list value types, then *BVR* has group degree of reference.

- ii) Otherwise, the declared type of *BVR* is not a group list value type and

Case:

- 1) If the declared type of *BVR* is material, then the degree of reference of *BVR* is unconditional singleton.

- 2) Otherwise, the declared type of *BVR* is nullable and the degree of reference of *BVR* is conditional singleton.

- 12) Let *DEG* be the degree of reference of *BVR*. If *DEG* is not singleton, then *DEG* shall be effectively bounded group and the declared type of *BVR* shall be a group list value type.

- 13) If the binding graph pattern *GP* of *BVR* is defined, then the projected field type *PFT* of *BVR* is the field type whose name is *BVN* and whose value type is determined as follows.

Case:

- a) If *GP* declares an <element variable> equivalent to *BVN*, then

Case:

- i) If *DEG* is unconditional singleton, then the value type of *PFT* is a value type that is determined using an implementation-defined (IW011) mechanism such that it contains every reference value of a graph element to which *BV* can be bound by the evaluation of *GP*.
- ii) If *DEG* is conditional singleton, then the value type of *PFT* is a value type that is determined using an implementation-defined (IW011) mechanism such that it contains every reference value of a graph element to which *BV* can be bound by the evaluation of *GP* and the null value.
- iii) Otherwise, *DEG* is group and the value type of *PFT* is determined using an implementation-defined (IW011) mechanism such that:
 - 1) The value type of *PFT* is either a group list value type or a dynamic union type whose component types are group list value types.
 - 2) The value type of *PFT* contains every list of reference values to a graph element to which *BV* can be bound by the evaluation of *GP*.
- b) Otherwise, *GP* declares a <path variable> equivalent to *BVN* and the value type of *PFT* is the path value type.

General Rules

- 1) Let *F* be the field of the current working record whose name is *BVN*.
- 2) Let *FV* be the value of *F*.
- 3) If the degree of reference of *BVR* is group and *FV* is not either a list value or the null value, then an exception condition is raised: *data exception — invalid group variable value (22G13)*.
- 4) The result of *BVR* is *FV*.

NOTE 327 — Every <binding variable reference> is evaluated by looking it up in the current working record. The referenced binding variable of a <binding variable reference> declared by a <graph pattern> is bound in the Subclause 22.6, “Application of bindings to evaluate an expression”.

Conformance Rules

- 1) Without Feature GP11, “Procedure-local graph variable definitions”, in conforming GQL language, the declared type of a <binding variable reference> shall not be a supertype of a graph reference value type.
- 2) Without Feature GP08, “Procedure-local binding table variable definitions”, in conforming GQL language, the declared type of a <binding variable reference> shall not be a supertype of a binding table reference value type.

20.13 <path value expression>

Function

Specify a path value.

Format

```
<path value expression> ::=  
    <path value concatenation>  
    | <path value primary>  
  
<path value concatenation> ::=  
    <path value expression 1> <concatenation operator> <path value primary>  
  
<path value expression 1> ::=  
    <path value expression>  
  
<path value primary> ::=  
    <value expression primary>
```

Syntax Rules

- 1) The declared type of a <path value concatenation> is the path value type.
- 2) The declared type of a <path value primary> shall be a path value type.

General Rules

- 1) If <path value concatenation> is specified, then let *PV1* be the result of <path value expression 1> and let *PV2* be the result of <path value primary>.
Case:
 - a) If either *PV1* or *PV2* is the null value, then the result of the <path value concatenation> is the null value.
 - b) Otherwise:
 - i) Let *PEL1* and *PEL2* be the path element lists of *PV1* and *PV2*, respectively.
 - ii) If the last element of *PEL1* and the first element of *PEL2* are not node reference values whose referent is the same node, then an exception condition is raised: *data exception — malformed path (22G0Z)*.
 - iii) Let *PELF* be *PEL1* without its last element.
NOTE 328 — If *PEL1* is a single-node path value, then *PELF* is the empty sequence.
 - iv) Let *IDMC* be the implementation-defined (IL015) maximum cardinality of path element lists of path value types. If the sum of the cardinality of *PELF* and the cardinality of *PEL2* is greater than *IDMC*, then an exception condition is raised: *data exception — path data, right truncation (22G10)*.
 - v) The result of the <path value concatenation> is the path value whose path element list comprises every element of *PELF* followed by every element of *PEL2*.

Conformance Rules

- 1) Without Feature GV55, “Path value types”, conforming GQL language shall not contain a <path value expression>.
- 2) Without Feature GE06, “Path value construction”, conforming GQL language shall not contain a <path value concatenation>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

20.14 <path value constructor>

Function

Specify construction of a path value.

Format

```
<path value constructor> ::=  
  <path value constructor by enumeration>  
  
<path value constructor by enumeration> ::=  
  PATH <left bracket> <path element list> <right bracket>  
  
<path element list> ::=  
  <path element list start> [ <path element list step>... ]  
  
<path element list start> ::=  
  <node reference value expression>  
  
<path element list step> ::=  
  <comma> <edge reference value expression> <comma> <node reference value expression>
```

Syntax Rules

- 1) The declared type of the <path value constructor by enumeration> is the path value type.

General Rules

- 1) If the <path value constructor by enumeration> *PVCBE* is specified, then:
 - a) Let *PEL* be the list of results of all <node reference value expression>s and <edge reference value expression>s simply contained in *PVCBE*.
 - b) If *PEL* contains the null value or does not identify a path, then an exception condition is raised: *data exception — malformed path (22G0Z)*
 - c) The result of *PVCBE* is the path value whose path element list is *PEL*.

Conformance Rules

- 1) Without Feature GE06, “Path value construction”, conforming GQL language shall not contain a <path value constructor>.

20.15 <list value expression>

Function

Specify a list value.

Format

```
<list value expression> ::=  
    <list concatenation>  
    | <list primary>  
  
<list concatenation> ::=  
    <list value expression 1> <concatenation operator> <list primary>  
  
<list value expression 1> ::=  
    <list value expression>  
  
<list primary> ::=  
    <list value function>  
    | <value expression primary>
```

Syntax Rules

- 1) The declared type of a <list primary> shall be a list type.
- 2) If <list concatenation> is specified, then:
 - a) The Syntax Rules of [Subclause 22.18](#), “General combination of value types”, are applied with the declared types of <list value expression 1> and <list primary> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those Syntax Rules.
 - b) Let *IDMC* be the implementation-defined ([IL015](#)) maximum cardinality of a list type.
 - c) The declared type of <list concatenation> is the regular variant of *DT*.

General Rules

- 1) If <list concatenation> is specified, then let *LV1* be the result of <list value expression 1> and let *LV2* be the value of <list primary>.
Case:
 - a) If at least one of *LV1* and *LV2* is the null value, then the result of the <list concatenation> is the null value.
 - b) If the sum of the cardinality of *LV1* and the cardinality of *LV2* is greater than *IDMC*, then an exception condition is raised: *data exception — list data, right truncation (22G0B)*.
 - c) Otherwise, the result is the list comprising every element of *LV1* followed by every element of *LV2*.

Conformance Rules

- 1) Without Feature GV50, “List value types”, conforming GQL language shall not contain a <list value expression>.

20.16 <list value function>

Function

Specify a function yielding a value of a list type.

Format

```
<list value function> ::=  
    <trim list function>  
    | <elements function>  
  
<trim list function> ::=  
    TRIM <left paren> <list value expression> <comma> <numeric value expression> <right paren>  
  
<elements function> ::=  
    ELEMENTS <left paren> <path value expression> <right paren>
```

Syntax Rules

- 1) If <trim list function> is specified, then:
 - a) The declared type of the <numeric value expression> shall be an exact numeric type with scale 0 (zero).
 - b) The declared type of the <trim list function> is the regular variant of the declared type of the immediately contained <list value expression>.
- 2) If <elements function> is specified, then the declared type of <elements function> is an implementation-defined (ID005) list value type that includes all node and edge reference values.

General Rules

- 1) The result of <trim list function> is defined as follows:
 - a) Let *NV* be the result of the <numeric value expression>.
 - b) If *NV* is the null value, then the result is the null value and no further General Rules of this Subclause are applied.
 - c) If *NV* is less than 0 (zero), then an exception condition is raised: *data exception — list element error* (22G0C).
 - d) Let *AV* be the result of the <list value expression>.
 - e) If *AV* is the null value, then the result is the null value and no further General Rules of this Subclause are applied.
 - f) Let *AC* be the cardinality of *AV*.
 - g) If *NV* is greater than *AC*, then an exception condition is raised: *data exception — list element error* (22G0C).
 - h) Let *N* be *AC* – *NV*.
 - i) Case:
 - i) If *N* = 0 (zero), then the result is a list whose cardinality is 0 (zero).

- ii) Otherwise, the result is a list of N elements such that, for i , $1 \text{ (one)} \leq i \leq N$, the value of the i -th element of the result is the value of the i -th element of AV .
- 2) The result of <elements function> EF is defined as follows.
- a) Let PV be the result of the <path value expression>.
 - b) If PV is the null value, then the result of EF is the null value; otherwise, the result of EF is the path element list of PV .

Conformance Rules

- 1) Without Feature GV50, “List value types”, conforming GQL language shall not contain a <list value function>.
- 2) Without Feature GF04, “Enhanced path functions”, conforming GQL language shall not contain an <elements function>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

20.17 <list value constructor>

Function

Specify construction of a list.

Format

```
<list value constructor> ::=  
  <list value constructor by enumeration>  
  
<list value constructor by enumeration> ::=  
  [ <list value type name> ] <left bracket> [ <list element list> ] <right bracket>  
  
<list element list> ::=  
  <list element> [ { <comma> <list element> }... ]  
  
<list element> ::=  
  <value expression>
```

Syntax Rules

- 1) The declared type of <list value constructor> is the declared type of the immediately contained <list value constructor by enumeration>.
- 2) Let *LVCE* be the <list value constructor by enumeration>. Let *C* be the number of <list element>s simply contained in *LVCE*.

NOTE 329 — If <list value constructor by enumeration> does not simply contain a <list element list>, then *C* is equal to 0 (zero).
- 3) Case:
 - a) If *C* is greater than 0 (zero), then the Syntax Rules of [Subclause 22.18](#), “General combination of value types”, are applied with the declared types of the <list element>s immediately contained in the <list element list> of *LVCE* as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those Syntax Rules. The declared type of *LVCE* is the regular list value type with element type *DT* and a maximum cardinality *C*.
 - b) Otherwise, *C* is equal to 0 (zero) and
 - i) If the GQL-implementation supports Feature GV72, “Immaterial value types: empty type support”, then the declared type of *LVCE* is the list value type whose list element type is the empty type.
 - ii) Otherwise, the declared type of *LVCE* is a list value type with an implementation-defined (ID004) list element type and a maximum cardinality of *C*.
 - c) *C* shall be less or equal than the implementation-defined (IL015) maximum cardinality for list value types whose list element type is *DT*.

General Rules

- 1) The result of <list value constructor by enumeration> is a list value whose *i*-th element is the value of the *i*-th <list element> simply contained in the <list value constructor by enumeration>, cast as the value type of *DT*.

ISO/IEC 39075:2024(en)
20.17 <list value constructor>

NOTE 330 — If <list value constructor by enumeration> does not simply contain a <list element list>, then the result is a list value that is an empty list value.

Conformance Rules

- 1) Without Feature GV50, “List value types”, conforming GQL language shall not contain a <list value constructor>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

20.18 <record constructor>

Function

Specify construction of a record.

Format

```
<record constructor> ::=  
  [ RECORD ] <fields specification>  
  
<fields specification> ::=  
  <left brace> [ <field list> ] <right brace>  
  
<field list> ::=  
  <field> [ { <comma> <field> }... ]
```

Syntax Rules

- 1) Let *RVC* be the <record constructor>.
- 2) Every <fields specification> specifies the fields specified by the immediately contained <field list>.
- 3) Every <field list> *FL* specifies the fields defined by the <field>s simply contained in *FL*.
- 4) Let *FS* be the <fields specification> immediately contained in *RVC*. Let *N* be the number of fields specified by *FS*. *N* shall be less or equal than the implementation-defined (IL015) maximum number of record fields. For *i*, 1 (one) $\leq i \leq N$, let *F_i* be the *i*-th field specified by *FS*, and let *FT_i* be the field type of *F_i*.
- 5) Let *RT* be defined as follows:
 - a) Let *RT₀* be defined as follows:
 - i) If *N* is 0 (zero), then *RT₀* is:
RECORD { }
 - ii) Otherwise, *RT₀* is
RECORD { *FT₁*, ..., *FT_N* }
 - b) *RT* is the <record type> obtained by the application of Syntax Rules for <record type> to *RT₀*.
- 6) The declared type of *RVC* is the record type specified by *RT*.

General Rules

- 1) The General Rules of <record type> are applied to *RT*.
- 2) The value of *RVC* is the record whose set of fields are the fields defined by the <field>s specified by *FS*.

Conformance Rules

- 1) Without Feature GV45, “Record types”, conforming GQL language shall not contain a <record constructor>.
- 2) Without Feature GV46, “Closed record types”, in conforming GQL language, the declared type of the <record constructor> is the open record type.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

20.19 <field>

Function

Specify a field.

Format

```
<field> ::=  
  <field name> <colon> <value expression>
```

Syntax Rules

- 1) Let *FL* be the <field list> that simply contains a <field> *F*.
- 2) The <field name> shall not be equivalent to the <field name> of any other <field> simply contained in *FL*.
- 3) The <field name> specifies the *field name* of the field defined by *F*.
- 4) The <value expression> specifies the *field value expression* of *F*.
- 5) The field type of *F* is the pair comprising the field name of the field defined by *F* and the declared type of the field value expression of *F*.

General Rules

- 1) The *field value* of the field defined by *F* is the value of the field value expression of *F*.

Conformance Rules

- 1) Without Feature GV48, “Nested record types”, conforming GQL language shall not contain a <field> that simply contains a <record constructor>.

20.20 <boolean value expression>

Function

Specify a Boolean value.

Format

```
<boolean value expression> ::=  
    <boolean term>  
    | <boolean value expression> OR <boolean term>  
    | <boolean value expression> XOR <boolean term>  
  
<boolean term> ::=  
    <boolean factor>  
    | <boolean term> AND <boolean factor>  
  
<boolean factor> ::=  
    [ NOT ] <boolean test>  
  
<boolean test> ::=  
    <boolean primary> [ IS [ NOT ] <truth value> ]  
  
<truth value> ::=  
    TRUE  
    | FALSE  
    | UNKNOWN  
  
<boolean primary> ::=  
    <predicate>  
    | <boolean predicand>  
  
<boolean predicand> ::=  
    <parenthesized boolean value expression>  
    | <non-parenthesized value expression primary>  
  
<parenthesized boolean value expression> ::=  
    <left paren> <boolean value expression> <right paren>
```

Syntax Rules

- 1) The declared type of a <boolean value expression> is a Boolean type.
- 2) The declared type of a <parenthesized boolean value expression> is the declared type of the simply contained <boolean value expression>.
- 3) The declared type of a <boolean predicand> that is a <non-parenthesized value expression primary> shall be a Boolean type.
- 4) $X \text{ XOR } Y$ is equivalent to the following <boolean value expression>:
 $(X \text{ OR } Y) \text{ AND NOT } (X \text{ AND } Y)$
- 5) 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:
 $(\text{NOT } (BP \text{ IS } TV))$

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 5, “Truth table for the AND Boolean operator”, Table 6, “Truth table for the OR Boolean operator”, and Table 7, “Truth table for the IS Boolean operator”, specify the semantics of AND, OR, and IS, respectively.

Table 5 — Truth table for the AND Boolean operator

AND	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Unknown</i>	<i>Unknown</i>	<i>False</i>	<i>Unknown</i>

Table 6 — Truth table for the OR Boolean operator

OR	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>Unknown</i>	<i>True</i>	<i>Unknown</i>	<i>Unknown</i>

Table 7 — Truth table for the IS Boolean operator

IS	TRUE	FALSE	UNKNOWN
<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>
<i>Unknown</i>	<i>False</i>	<i>False</i>	<i>True</i>

Conformance Rules

- 1) Without Feature GE07, “Boolean XOR”, conforming GQL language shall not contain a <boolean value expression> that immediately contains XOR.

20.21 <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) Case:
 - a) If the declared type of either operand of a dyadic arithmetic operator is approximate numeric, then the declared type of the result is an implementation-defined (ID063) approximate numeric type.
 - b) Otherwise, the declared type of both operands of a dyadic arithmetic operator is exact numeric and the declared type of the result is an implementation-defined (ID064) exact numeric type, with precision and scale defined as follows:
 - i) Let $S1$ and $S2$ be the scale of the first and second operands respectively.
 - ii) The precision of the result of addition and subtraction is implementation-defined (ID065), and the scale is the maximum of $S1$ and $S2$.
 - iii) The precision of the result of multiplication is implementation-defined (ID066), and the scale is $S1 + S2$.
 - iv) The precision and scale of the result of division are implementation-defined (ID067).
- 2) The declared type of a <factor> is that of the immediately contained <numeric primary>.
- 3) The declared type of a <numeric primary> shall be numeric.
- 4) If a <numeric value expression> immediately contains a <minus sign> *NMS* and immediately contains a <term> that is a <factor> that immediately contains a <sign> that is a <minus sign> *FMS*, then there shall be a <separator> between *NMS* and *FMS*.

General Rules

- 1) If the result 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 result 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

Case:

- a) If a GQL-implementation supports Feature GA01, “IEEE 754 floating point operations”, then the result shall be a value as defined by IEEE Std 754:2019.
 - b) Otherwise, an exception condition is raised: *data exception — division by zero (22012)*.
- 5) If the result of an arithmetic operation is an exact number, 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 declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.
 - b) If the operator is division and the approximate mathematical result of the operation represented with the precision and scale of the declared type of the result 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 (22003)*. The choice of whether to round or truncate is implementation-defined (IA011).
- 6) If the result of an arithmetic operation is an approximate number and the exponent of the approximate mathematical result of the operation is not within the implementation-defined (IL023) exponent range for the declared type of the result, then

Case:

- a) If a GQL-implementation supports Feature GA01, “IEEE 754 floating point operations”, then the result shall be a value as defined by IEEE Std 754:2019.
- b) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

Conformance Rules

None.

20.22 <numeric value function>

Function

Specify a function yielding a value of a numeric type.

Format

```
<numeric value function> ::=
  <length expression>
  | <cardinality expression>
  | <absolute value expression>
  | <modulus expression>
  | <trigonometric function>
  | <general logarithm function>
  | <common logarithm>
  | <natural logarithm>
  | <exponential function>
  | <power function>
  | <square root>
  | <floor function>
  | <ceiling function>

<length expression> ::=
  <char length expression>
  | <byte length expression>
  | <path length expression>

<cardinality expression> ::=
  CARDINALITY <left paren> <cardinality expression argument> <right paren>
  | SIZE <left paren> <list value expression> <right paren>

<cardinality expression argument> ::=
  <binding table reference value expression>
  | <path value expression>
  | <list value expression>
  | <record expression>

<char length expression> ::=
  { CHAR_LENGTH | CHARACTER_LENGTH }
  <left paren> <character string value expression> <right paren>

<byte length expression> ::=
  { BYTE_LENGTH | OCTET_LENGTH }
  <left paren> <byte string value expression> <right paren>

<path length expression> ::=
  PATH_LENGTH <left paren> <path 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>
```

ISO/IEC 39075:2024(en)
20.22 <numeric value function>

<trigonometric function> ::=
 <trigonometric function name> <left paren> <numeric value expression> <right paren>

<trigonometric function name> ::=
 SIN | COS | TAN | COT | SINH | COSH | TANH | ASIN | ACOS | ATAN | DEGREES | RADIANS

<general logarithm function> ::=
 LOG <left paren> <general logarithm base> <comma>
 <general logarithm argument> <right paren>

<general logarithm base> ::=
 <numeric value expression>

<general logarithm argument> ::=
 <numeric value expression>

<common logarithm> ::=
 LOG10 <left paren> <numeric value expression> <right paren>

<natural logarithm> ::=
 LN <left paren> <numeric value expression> <right paren>

<exponential function> ::=
 EXP <left paren> <numeric value expression> <right paren>

<power function> ::=
 POWER <left paren> <numeric value expression base> <comma>
 <numeric value expression exponent> <right paren>

<numeric value expression base> ::=
 <numeric value expression>

<numeric value expression exponent> ::=
 <numeric value expression>

<square root> ::=
 SQRT <left paren> <numeric value expression> <right paren>

<floor function> ::=
 FLOOR <left paren> <numeric value expression> <right paren>

<ceiling function> ::=
 { CEIL | CEILING } <left paren> <numeric value expression> <right paren>

Syntax Rules

- 1) The declared type of the result of <length expression> is the implementation-defined (ID068) nullable exact numeric type with scale 0 (zero).
- 2) If a <cardinality expression> *CE* is specified that immediately contains SIZE and the <list value expression> *LVE*, then *CE* is effectively replaced by:

 CARDINALITY(*LVE*)
- 3) The declared type *DT* of the result of a <cardinality expression> *CE* that immediately contains a <cardinality expression argument> *CEA* is defined as follows:
 - a) Let *DTA* be the declared type of *CEA*.
 - b) *DT* is the implementation-defined (ID070) exact numeric type with scale 0 (zero) to whose nullability the nullability of *DTA* is assignment-aligned.

- 4) The declared type of the result of <absolute value expression> the declared type of the immediately contained <numeric value expression>.
- 5) The declared type of the result of <modulus expression> is the declared type of the immediately contained <numeric value expression divisor>.
- 6) The declared type of the result of <trigonometric function> is an implementation-defined (ID069) approximate numeric type.
- 7) The declared type of the result of <general logarithm function> is an implementation-defined (ID069) approximate numeric type.
- 8) The declared type of the result of <natural logarithm> is an implementation-defined (ID069) approximate numeric type.
- 9) The declared type of the result of <exponential function> is an implementation-defined (ID069) approximate numeric type.
- 10) The declared type of the result of <power function> is an implementation-defined (ID069) approximate numeric type.
- 11) If <floor function> or <ceiling function> is specified, then
Case:
 - a) If the declared type of the immediately contained <numeric value expression> *NVE* is exact numeric, then the declared type of the result is exact numeric with implementation-defined (ID074) precision, with the radix of *NVE*, and with scale 0 (zero).
 - b) Otherwise, an approximate numeric with implementation-defined (ID075) precision.
- 12) If <common logarithm> is specified, then let *NVE* be the simply contained <numeric value expression>. The <common logarithm> is equivalent to:
 $\text{LOG} (10, NVE)$
- 13) If <square root> is specified, then let *NVE* be the simply contained <numeric value expression>. The <square root> is equivalent to:
 $\text{POWER} (NVE, 0.5)$

General Rules

- 1) If the <char length expression> *CLX* is specified, then let the character string *CSV* be the result of the <character string value expression> immediately contained in *CLX*.
Case:
 - a) If *CSV* is material, then the result of *CLX* is the (character string) length of *CSV*.

NOTE 331 — The (character string) length of *CSV* is defined in Subclause 4.16.3, “Character string types” as the number of characters in the sequence of characters comprising *CSV*.
 - b) Otherwise, the result of *CLX* is the null value.
- 2) If the <byte length expression> *BLX* is specified, then let the byte string *BSV* be the result of the <byte string value expression> immediately contained in *BLX*.
Case:
 - a) If *BSV* is material, then the result of *BLX* is the (byte string) length of *BSV*.

ISO/IEC 39075:2024(en)
20.22 <numeric value function>

NOTE 332 — The (byte string) length of *BSV* is defined in Subclause 4.16.4, “Byte string types” as the number of bytes in the sequence of bytes comprising *BSV*.

b) Otherwise, the result of *BLX* is the null value.

- 3) If the <path length expression> *PLX* is specified, then let the path value *PSV* be the result of the <path value expression> immediately contained in *PLX*.

Case:

- a) If *PSV* is material, then the result of *PLX* is the (path value) length of *PSV*.

NOTE 333 — The (path value) length of *PSV* is defined in Subclause 4.15.2, “Path value types” as the number of edge reference values in the path element list *PEL* of *PSV*, if *PEL* contains at least one node reference value; otherwise, the null value.

b) Otherwise, the result of *PLX* is the null value.

- 4) If the <cardinality expression> *CE* is specified, then let *CER* be the result of the <cardinality expression argument> immediately contained in *CE* and the result of *CE* is defined as follows.

Case:

- a) If *CER* is a binding table reference value whose referent is *BT*, then the result of *CE* is the cardinality of *BT*.
- b) If *CER* is a path value *PV*, then the result of *CE* is the cardinality of *PV*.
- c) If *CER* is a list value *LV*, then the result of *CE* is the cardinality of *LV*.
- d) If *CER* is a record *REC*, then the result of *CE* is the cardinality of *REC*.
- e) Otherwise, *CER* is the null value and the result of *CE* is the null value.

- 5) If <absolute value expression> is specified, then let *N* be the result 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 declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

- 6) If <modulus expression> is specified, then let *N* be the result of the immediately contained <numeric value expression dividend> and let *M* be the result of the immediately contained <numeric value expression divisor>.

Case:

- a) If at least one of *N* and *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 (22012)*.
- c) Otherwise, the result is the unique exact numeric value *R* with scale 0 (zero) such that all of the following are true:
- i) *R* has the same sign as *N*.
 - ii) The absolute value of *R* is less than the absolute value of *M*.
 - iii) For some exact numeric value *K* with scale 0 (zero), $N = M * K + R$.

- 7) If <trigonometric function> is specified, then let V be the result of the immediately contained <numeric value expression> that represents an angle expressed in radians.

Case:

- a) If V is the null value, then the result is the null value.
- b) Otherwise, let OP be the <trigonometric function name>.

Case:

- i) If OP is ACOS, then

Case:

- 1) If V is less than -1 (negative one) or V is greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- 2) Otherwise, the result of the <trigonometric function> is the inverse cosine of V .

- ii) If OP is ASIN, then

Case:

- 1) If V is less than -1 (negative one) or V is greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- 2) Otherwise, the result of the <trigonometric function> is the inverse sine of V .

- iii) If OP is ATAN, then the result is the inverse tangent of V .

- iv) If OP is COS, then the result is the cosine of V .

- v) If OP is COSH, then the result is the hyperbolic cosine of V .

- vi) If OP is SIN, then the result is the sine of V .

- vii) If OP is SINH, then the result is the hyperbolic sine of V .

- viii) If OP is TAN, then the result is the tangent of V .

- ix) If OP is TANH, then the result is the hyperbolic tangent of V .

- x) If OP is COT, then the result is the cotangent of V .

- xi) If OP is DEGREES, then the result is the value of V , taken to be in radians, expressed as degrees.

- xii) If OP is RADIANS, then the result is the value of V , taken to be in degrees, expressed as radians.

- 8) If <general logarithm function> is specified, then let VB be the result of the <general logarithm base> and let VA be the result of the <general logarithm argument>.

Case:

- a) If at least one of VA and VB is the null value, then the result is the null value.
- b) If VA is negative or 0 (zero), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- c) If VB is negative, 0 (zero), or 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- d) Otherwise, the result is the logarithm with base VB of VA .

- 9) If <natural logarithm> is specified, then let V be the result of the immediately contained <numeric value expression>.

Case:

- a) If V is the null value, then the result is the null value.
- b) If V is 0 (zero) or negative, then an exception condition is raised: *data exception — invalid argument for natural logarithm (2201E)*.
- c) Otherwise, the result is the natural logarithm of V .

- 10) If <exponential function> is specified, then let V be the result of the immediately contained <numeric value expression>.

Case:

- a) If V is the null value, then the result is the null value.
- b) Otherwise, the result is e (the base of natural logarithms) raised to the power V . If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

- 11) If <power function> is specified, then let $NVEB$ be the <numeric value expression base>, then let VB be the result of $NVEB$, let $NVEE$ be the <numeric value expression exponent>, and let VE be the result of $NVEE$.

Case:

- a) If at least one of VB and VE is the null value, then the result is the null value.
- b) If VB is 0 (zero) and VE is negative, then an exception condition is raised: *data exception — invalid argument for power function (2201F)*.
- c) If VB is 0 (zero) and VE is 0 (zero), then the result is 1 (one).
- d) If VB is 0 (zero) and VE is positive, then the result is 0 (zero).
- e) If VB is negative and VE is not equal to an exact numeric value with scale 0 (zero), then an exception condition is raised: *data exception — invalid argument for power function (2201F)*.
- f) If VB is negative and VE is equal to an exact numeric value with scale 0 (zero) that is an even number, then the result is the result of

$$\text{EXP}(NVEE * \text{LN}(-NVEB))$$

- g) If VB is negative and VE is equal to an exact numeric value with scale 0 (zero) that is an odd number, then the result is the result of

$$-\text{EXP}(NVEE * \text{LN}(-NVEB))$$

- h) Otherwise, the result is the result of

$$\text{EXP}(NVEE * \text{LN}(NVEB))$$

- 12) If <floor function> is specified, then let V be the result of the immediately contained <numeric value expression> NVE .

Case:

- a) If V is the null value, then the result is the null value.
- b) Otherwise,

Case:

- i) If the declared type of *NVE* is exact numeric, then the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- ii) Otherwise, the result is the greatest whole number that is less than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

- 13) If <ceiling function> is specified, then let *V* be the result of the immediately contained <numeric value expression> *NVE*.

Case:

- a) If *V* is the null value, then the result is the null value.
- b) Otherwise,

Case:

- i) If the declared type of *NVE* is exact numeric, then the result is the least exact numeric value with scale 0 (zero) that is greater than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- ii) Otherwise, the result is the least whole number that is greater than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

Conformance Rules

- 1) Without Feature GF01, “Enhanced numeric functions”, conforming GQL language shall not contain an <absolute value expression>, a <modulus expression>, a <floor function>, a <ceiling function>, or a <square root>.
- 2) Without Feature GF02, “Trigonometric functions”, conforming GQL language shall not contain a <trigonometric function>.
- 3) Without Feature GF03, “Logarithmic functions”, conforming GQL language shall not contain a <general logarithm function>, a <common logarithm>, a <natural logarithm>, an <exponential function>, or a <power function>.
- 4) Without Feature GV35, “Byte string types”, conforming GQL language shall not contain a <byte length expression>.
- 5) Without Feature GF04, “Enhanced path functions”, conforming GQL language shall not contain a <path length expression>.
- 6) Without Feature GF12, “CARDINALITY function”, conforming GQL language shall not contain a <cardinality expression> that immediately contains CARDINALITY.
- 7) Without Feature GF13, “SIZE function”, conforming GQL language shall not contain a <cardinality expression> that immediately contains SIZE.

20.23 <string value expression>

Function

Specify a character string value or a byte string value.

Format

```
<string value expression> ::=  
    <character string value expression>  
    | <byte string value expression>  
  
<character string value expression> ::=  
    <character string concatenation>  
    | <character string primary>  
  
<character string concatenation> ::=  
    <character string value expression> <concatenation operator> <character string primary>  
  
<character string primary> ::=  
    <value expression primary>  
    | <character string function>  
  
<byte string value expression> ::=  
    <byte string concatenation>  
    | <byte string primary>  
  
<byte string primary> ::=  
    <value expression primary>  
    | <byte string function>  
  
<byte string concatenation> ::=  
    <byte string value expression> <concatenation operator> <byte string primary>
```

Syntax Rules

- 1) If a <character string concatenation> *CSC* is specified, then let *CSVE* be the <character string value expression> immediately contained in *CSC* and let *CSF* be the <character string primary> immediately contained in *CSC*. Let *C1* be the declared type of *CSVE* and let *C2* be the declared type of *CSF*. Let *M* be the length in characters of *C1* plus the length in characters of *C2*. Let *VL* be the implementation-defined (IL013) maximum length of character strings.
Case:
 - a) If either *C1* or *C2* is a variable-length character string type, then the declared type *DTCS* of *CSC* is a variable-length character string. The minimum length of *DTCS* is an implementation-defined (IL009) non-negative exact number that is less than or equal to the sum of the minimum lengths of *C1* and *C2*, and the maximum length of *DTCS* is equal to the lesser of *M* and *VL*.
 - b) If both *C1* and *C2* are fixed-length character string types, then declared type *DTCS* of *CSC* is a fixed-length character string type. The minimum and maximum lengths of *DTCS* are equal to *M* and *M* shall not be greater than *VL*.
- 2) The declared type of a <character string primary> shall be a character string type.
- 3) The declared type *DTBSC* of a <byte string concatenation> *BSC* is a byte string type. Let *BSVE* be the <byte string value expression> immediately contained in *BSC* and let *BSF* be the <byte string primary> immediately contained in *BSC*. Let *B1* be the declared type of *BSVE* and let *B2* be the declared type

of *BSF*. Let *M* be the length in bytes of *B1* plus the length in bytes of *B2*. Let *VL* be the implementation-defined (IL013) maximum length of byte strings.

Case:

- a) If either *B1* or *B2* is a variable-length byte string type, then *DTBSC* is a variable-length byte string type. The minimum length of *DTBSC* is an implementation-defined (IL009) non-negative exact number that is less than or equal to the sum of the minimum lengths of *B1* and *B2*, and the maximum length of *DTBSC* is equal to the lesser of *M* and *VL*.
 - b) If both *B1* and *B2* are fixed-length byte string types, then *DTBSC* is a fixed-length byte string type. The minimum and maximum lengths of *DTBSC* are equal to *M* and *M* shall not be greater than *VL*.
- 4) The declared type of a <byte string primary> shall be a byte string type.

General Rules

- 1) If the result of any <character string primary> or <byte string 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 <character string concatenation> is specified, then:
 - a) Let *S1* and *S2* be the result of the <character string value expression> and <character string primary>, respectively.

Case:

- i) If at least one of *S1* and *S2* is the null value, then the result of the <character string concatenation> is the null value.
- ii) Otherwise:
 - 1) Let *S* be the character string comprising *S1* followed by *S2* and let *M* be the length of *S*.
 - 2) *S* is effectively replaced by

Case:

- A) If the <search condition> *S1* IS NORMALIZED AND *S2* IS NORMALIZED evaluates to True, then:

NORMALIZE (*S*)

- B) Otherwise, an implementation-defined (IW017) character string.

- 3) Let *VL* be the implementation-defined (IL013) maximum length of character strings.

Case:

- A) If *M* is less than or equal to *VL*, then the result of the <character string concatenation> is *S* with length *M*.
- B) If *M* is greater than *VL* and the right-most *M*−*VL* characters of *S* are all <truncating whitespace> characters, then the result of the <character string concatenation> is the first *VL* characters of *S* with length *VL*.
- C) Otherwise, an exception condition is raised: *data exception — string data, right truncation* (22001).

- 3) If <byte string concatenation> is specified, then let $S1$ and $S2$ be the result of the <byte string value expression> $SE1$ and <byte string primary> $SE2$, respectively.

Case:

- a) If at least one of $S1$ and $S2$ is the null value, then the result of the <byte string concatenation> is the null value.
- b) Otherwise, let S be the byte string comprising $S1$ followed by $S2$; and let M be the length in bytes of S .

Case:

- i) If the declared type of at least one of $SE1$ and $SE2$ is a variable-length byte string type, then let VL be the implementation-defined (IL013) maximum length of byte strings.

Case:

- 1) If M is less than or equal to VL , then the result of the <byte string concatenation> is S with length M .
 - 2) If M is greater than VL and the right-most $M-VL$ bytes of S are all X'00', then the result of the <byte string concatenation> is the first VL bytes of S with length VL .
 - 3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.
- ii) Otherwise, the declared types of both $SE1$ and $SE2$ are fixed-length byte string types, and the result of the <byte string concatenation> is S .

Conformance Rules

None.

20.24 <character string function>

Function

Specify a function yielding a character string value.

Format

```
<character string function> ::=
    <substring function>
    | <fold>
    | <trim function>
    | <normalize function>

<substring function> ::=
    { LEFT | RIGHT }
    <left paren> <character string value expression> <comma> <string length> <right paren>

<fold> ::=
    { UPPER | LOWER } <left paren> <character string value expression> <right paren>

<trim function> ::=
    <single-character trim function>
    | <multi-character trim function>

<single-character trim function> ::=
    TRIM <left paren> <trim operands> <right paren>

<multi-character trim function> ::=
    { BTRIM | LTRIM | RTRIM }
    <left paren> <trim source> [ <comma> <trim character string> ] <right paren>

<trim operands> ::=
    [ [ <trim specification> ] [ <trim character string> ] FROM ] <trim source>

<trim source> ::=
    <character string value expression>

<trim specification> ::=
    LEADING
    | TRAILING
    | BOTH

<trim character string> ::=
    <character string value expression>

<normalize function> ::=
    NORMALIZE <left paren> <character string value expression>
    [ <comma> <normal form> ] <right paren>

<normal form> ::=
    NFC
    | NFD
    | NFKC
    | NFKD

<string length> ::=
    <numeric value expression>
```

Syntax Rules

- 1) If <substring function> is specified, then the declared type of the result is the declared type of the immediately contained <character string value expression>.
- 2) If <fold> is specified, then the declared type of the result is the declared type of the immediately contained <character string value expression>.
- 3) If the <single-character trim function> *SCTF* is specified, then:
 - a) The declared type of the result is the declared type of the <trim source> simply contained in *SCTF*.
 - b) Case:
 - i) If FROM is specified, then:
 - 1) At least one of <trim specification> and <trim character string> shall be specified.
 - 2) If a <trim character string> is specified, then the declared type of <trim character string> and the declared type of <trim source> shall be comparable value types.
 - 3) If <trim specification> is not specified, then BOTH is implicit.
 - 4) If <trim character string> is not specified, then <space> is implicit.
 - ii) Otherwise, let *SRC* be <trim source>. *TRIM* (*SRC*) is equivalent to:
TRIM (BOTH ' ' FROM *SRC*)
- 4) If the <multi-character trim function> *MCTF* is specified, then:
 - a) The declared type of the result is the declared type of the <trim source> *TS* immediately contained in *MCTF*.
 - b) Case:
 - i) If a <trim character string> *TC* is specified, then the declared type of *TS* and the declared type of *TC* shall be comparable value types.
 - ii) Otherwise, let *TC* be <space>.
 - c) If *BTRIM* is specified, then *BTRIM* (*TS*) is equivalent to:
BTRIM (*LTRIM* (*TS*))
- 5) If <normalize function> is specified, then
 - a) The declared type of the result is the declared type of the <character string value expression> simply contained in the <normalize function>.
 - b) Case:
 - i) If <normal form> is specified, then let *NF* be <normal form>.
 - ii) Otherwise, let *NF* be NFC.

General Rules

- 1) If <substring function> *SF* is specified, then:
 - a) Let *CSVE* be the result of the immediately contained <character string value expression>.

ISO/IEC 39075:2024(en)
20.24 <character string function>

- b) Let *SL* be the result of the immediately contained <string length>.
 - c) If at least one of *CSVE* and *SL* is the null value, then the result of *SF* is the null value and no further General Rules of this Subclause are applied.
 - d) If *SL* is less than 0 (zero), then an exception condition is raised: *data exception — substring error (22011)*.
 - e) Let *LC* be the length in characters of *CSVE*. If *SL* is greater than *LC*, then let *NC* be *LC*; otherwise, let *NC* be *SL*.
 - f) Case:
 - i) If *NC* is 0 (zero), then the result of *SF* is the zero-length character string.
 - ii) If LEFT is specified, then the result of *SF* is the character string containing the first *NC* characters of *CSVE*.
 - iii) If RIGHT is specified, then the result of *SF* is the character string containing the last *NC* characters of *CSVE*.
- 2) If <normalize function> is specified, then:
- a) Let *S* be the result of <character string value expression>.
 - b) If *S* is the null value, then the result of the <normalize function> is the null value.
 - c) Let *NR* be *S* in the normalization form specified by *NF* in accordance with [Unicode Standard Annex #15](#).
 - d) If the length in characters of *NR* is less than or equal to the implementation-defined (IL013) maximum length of a character string, then the result of the <normalize function> is *NR*; otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.
- 3) If <fold> is specified, then:
- a) Let *S* be the result of the <character string 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 let *FR* be a copy of *S* in which every lower-case character that has a corresponding upper-case character or characters and every title case character that has a corresponding upper-case character or characters is replaced by that upper-case character or characters.
 - ii) If LOWER is specified, then let *FR* be a copy of *S* in which every upper-case character that has a corresponding lower-case character or characters and every title case character that has a corresponding lower-case character or characters is replaced by that lower-case character or characters.
 - d) *FR* is effectively replaced by

Case:

 - i) If the <search condition> *S IS NORMALIZED* evaluated to True, then:
 NORMALIZE (*FR*)
 - ii) Otherwise, *FR*.

ISO/IEC 39075:2024(en)
20.24 <character string function>

- e) Let *FRL* be the length in characters of *FR* and let *FRML* be the implementation-defined (IL013) maximum length of a character string.
 - f) Case:
 - i) If *FRL* is less than or equal to *FRML*, then the result of the <fold> is *FR*.
 - ii) Otherwise, the result of the <fold> is the first *FRML* characters of *FR*. If any of the right-most (*FRL* – *FRML*) characters of *FR* are not <truncating whitespace> characters, then an exception condition is raised: *data exception — string data, right truncation (22001)*.
- 4) If the <single-character trim function> *SCTF* is specified, then:
- a) Let *S* be the result of the <trim source>.
 - b) Let *SC* be the result of the <trim character string>.
 - c) If at least one of *S* and *SC* is the null value, then the result of *SCTF* is the null value and no further General Rules of this Subclause are applied.
 - d) If the length in characters of *SC* is not 1 (one), then an exception condition is raised: *data exception — trim error (22027)*.
 - e) If *FROM* is specified and *SC* and *S* are not comparable values, then an exception condition is raised: *data exception — values not comparable (22G04)*.
 - f) Case:
 - i) If *BOTH* is specified or if no <trim specification> is specified, then the result of *SCTF* is the value of *S* with every leading or trailing character equal to *SC* removed.
 - ii) If *TRAILING* is specified, then the result of *SCTF* is the value of *S* with every trailing character equal to *SC* removed.
 - iii) If *LEADING* is specified, then the result of *SCTF* is the value of *S* with every leading character equal to *SC* removed.
- 5) If the <multi-character trim function> *MCTF* is specified, then:
- a) Let *S* be the result of the <trim source>.
 - b) Let *SC* be the result of the <trim character string>.
 - c) If at least one of *S* and *SC* is the null value, then the result of *MCTF* is the null value and no further General Rules of this Subclause are applied.
 - d) If *S* is the zero-length character string or *SC* is the zero-length character string, then the result of *MCTF* is *S* and no further General Rules of this Subclause are applied.
 - e) If *SC* and *S* are not comparable values, then an exception condition is raised: *data exception — values not comparable (22G04)*.
 - f) Case:
 - i) If *RTRIM* is specified:
 - 1) Let *P* be the length in characters of *S*.
 - 2) Let *C* be the *P*-th character of *S*.
 - 3) While *C* is contained in *SC*, do:
 - A) Let *P* be *P* – 1 (one).

- B) Let C be the P -th character of S .
- 4) The result of $MCTF$ is a character string containing the first P characters of S .
- ii) If $LTRIM$ is specified:
 - 1) Let P be 1 (one).
 - 2) Let C be the P -th character of S .
 - 3) While C is contained in SC , do:
 - A) Let P be $P + 1$ (one).
 - B) Let C be the P -th character of S .
 - 4) The result of $MCTF$ is a character string containing the last P characters of S .

Conformance Rules

- 1) Without Feature GF05, “Multi-character TRIM function”, conforming GQL language shall not contain a <multi-character trim function>.
- 2) Without Feature GF06, “Explicit TRIM function”, conforming GQL language shall not contain a <single-character trim function> that simply contains FROM.

20.25 <byte string function>

Function

Specify a function yielding a byte string value.

Format

```
<byte string function> ::=  
    <byte string substring function>  
    | <byte string trim function>  
  
<byte string substring function> ::=  
    { LEFT | RIGHT }  
    <left paren> <byte string value expression> <comma> <string length> <right paren>  
  
<byte string trim function> ::=  
    TRIM <left paren> <byte string trim operands> <right paren>  
  
<byte string trim operands> ::=  
    [ [ <trim specification> ] [ <trim byte string> ] FROM ] <byte string trim source>  
  
<byte string trim source> ::=  
    <byte string value expression>  
  
<trim byte string> ::=  
    <byte string value expression>
```

Syntax Rules

- 1) If <byte string substring function> is specified, then the declared type of the result is a byte string type with maximum length equal to the maximum length of the declared type of the immediately contained <byte string value expression>.
- 2) If <byte string trim function> is specified, then the declared type of the result is a byte string type with maximum length equal to the maximum length of the declared type of the immediately contained <byte string value expression>.

General Rules

- 1) If <byte string substring function> *BSSF* is specified, then:
 - a) Let *BSVE* be the result of the immediately contained <byte string value expression>.
 - b) Let *BSL* be the result of the immediately contained <string length>.
 - c) If at least one of *BSVE* and *BSL* is the null value, then the result of *BSSF* is the null value and no further General Rules of this Subclause are applied.
 - d) If *BSL* is less than 0 (zero), then an exception condition is raised: *data exception — substring error (22011)*.
 - e) Let *BSLB* be the length in bytes of *BSVE*. If *BSL* is greater than *BSLB*, then let *NB* be *BSLB*; otherwise, let *NB* be *BSL*.
 - f) Case:
 - i) If *NB* is 0 (zero), then the result of *BSSF* is the zero-length byte string.

- ii) If LEFT is specified, then the result of *BSSF* is the byte string containing the first *NB* characters of *BSVE*.
 - iii) If RIGHT is specified, then the result of *BSSF* is the byte string containing the last *NB* characters of *BSVE*.
- 2) If <byte string trim function> *BSTF* is specified, then:
- a) Let *S* be the result of the <byte string trim source>.
 - b) Let *SO* be the result of <trim byte string>.
 - c) If at least one of *S* and *SO* is the null value, then the result of *BSTF* is the null value.
 - d) If the length in bytes of *SO* is not 1 (one), then an exception condition is raised: *data exception — trim error (22027)*.
 - e) Case:
 - i) If BOTH is specified or if no <trim specification> is specified, then the result of *BSTF* is the value of *S* with any leading or trailing bytes equal to *SO* removed.
 - ii) If TRAILING is specified, then the result of *BSTF* is the value of *S* with any trailing bytes equal to *SO* removed.
 - iii) If LEADING is specified, then the result of *BSTF* is the value of *S* with any leading bytes equal to *SO* removed.

Conformance Rules

- 1) Without Feature GF07, “Byte string TRIM function”, conforming GQL language shall not contain a <byte string trim function>.

20.26 <datetime value expression>

Function

Specify a datetime value.

Format

```
<datetime value expression> ::=  
    <datetime primary>  
    | <duration value expression> <plus sign> <datetime primary>  
    | <datetime value expression> <plus sign> <duration term>  
    | <datetime value expression> <minus sign> <duration term>  
  
<datetime primary> ::=  
    <value expression primary>  
    | <datetime value function>
```

Syntax Rules

- 1) If the declared type of the simply contained <datetime primary> is a zoned datetime type, then the declared type of <datetime value expression> is the zoned datetime type; otherwise, the declared type of <datetime value expression> is the local datetime type.

General Rules

- 1) If the result of a <datetime primary> is not a temporal instant, then an exception condition is raised: *data exception — invalid value type (22003)*.
- 2) If the result of any <datetime primary>, <duration value expression>, <datetime value expression>, or <duration term> simply contained in a <datetime value expression> is the null value, then the result of the <datetime value expression> is the null value.
- 3) The result of a <datetime primary> is the result of the immediately contained <value expression primary> or <datetime value function>.
- 4) If a <datetime value expression> immediately contains the operator <plus sign> or <minus sign>, then the result is evaluated as specified in ISO 8601-2:2019, 14, “Date and time arithmetic”.

Conformance Rules

- 1) Without Feature GV39, “Temporal types: date, local datetime and local time support”, conforming GQL language shall not contain a <datetime value expression> that contains a <localdatetime function> or a <localtime function>.
- 2) Without Feature GV40, “Temporal types: zoned datetime and zoned time support”, conforming GQL language shall not contain a <datetime value expression> that contains a <datetime function> or a <time function>.

20.27 <datetime value function>

Function

Specify a function yielding a value of a temporal instant type.

Format

```
<datetime value function> ::=
  <date function>
  | <time function>
  | <datetime function>
  | <localtime function>
  | <localdatetime function>

<date function> ::=
  CURRENT_DATE
  | DATE <left paren> [ <date function parameters> ] <right paren>

<time function> ::=
  CURRENT_TIME
  | ZONED_TIME <left paren> [ <time function parameters> ] <right paren>

<localtime function> ::=
  LOCAL_TIME [ <left paren> [ <time function parameters> ] <right paren> ]

<datetime function> ::=
  CURRENT_TIMESTAMP
  | ZONED_DATETIME <left paren> [ <datetime function parameters> ] <right paren>

<localdatetime function> ::=
  LOCAL_TIMESTAMP
  | LOCAL_DATETIME <left paren> [ <datetime function parameters> ] <right paren>

<date function parameters> ::=
  <date string>
  | <record constructor>

<time function parameters> ::=
  <time string>
  | <record constructor>

<datetime function parameters> ::=
  <datetime string>
  | <record constructor>
```

Syntax Rules

- 1) CURRENT_DATE is equivalent to:
DATE()
- 2) CURRENT_TIME is equivalent to:
ZONED_TIME()
- 3) LOCAL_TIME is equivalent to:
LOCAL_TIME()

- 4) CURRENT_TIMESTAMP is equivalent to:
ZONED_DATETIME()
- 5) LOCAL_TIMESTAMP is equivalent to:
LOCAL_DATETIME()
- 6) The declared type of a <date function> is the date type.
- 7) The declared type of a <time function> is the zoned time type.
- 8) The declared type of a <datetime function> is the zoned datetime type.
- 9) The declared type of a <localtime function> is the local time type.
- 10) The declared type of a <localdatetime function> is the local datetime type.

General Rules

- 1) If the current request timestamp is “not set”, then the current request timestamp is set to the current date and time at an implementation-dependent (US009) point in time before the evaluation of a <datetime value function>.

NOTE 334 — This can happen at any implementation-dependent (US009) point in time after the setting of the request timestamp to “not set”; the determination of the request timestamp need not wait for evaluation of a <datetime value function>.
- 2) If the set of <field name>s simply contained in a <date function parameters> is not one of:
 - a) 'year'
 - b) 'year' and 'month'
 - c) 'year', 'month', and 'day'then an exception is raised: *data exception — invalid date, time, or datetime function field name (22G05)*.
- 3) If the set of <field name>s simply contained in a <time function parameters> is not one of:
 - a) 'hour'
 - b) 'hour' and 'minute'
 - c) 'hour', 'minute', and 'second'
 - d) 'hour', 'minute', 'second' and one of 'millisecond', 'microsecond', and 'nanosecond'.
 - e) 'timezone', if <time function parameters> is simply contained in a <time function>.then an exception is raised: *data exception — invalid date, time, or datetime function field name (22G05)*.
- 4) If the set of <field name>s contained in a <datetime function parameters> is not one of:
 - a) 'year', 'month', 'day', and any of the set of <field name>s permitted in a <time function parameters>.
 - b) 'timezone', if <datetime function parameters> is simply contained in a <datetime function>.then an exception is raised: *data exception — invalid date, time, or datetime function field name (22G05)*.

ISO/IEC 39075:2024(en)
20.27 <datetime value function>

- 5) If the result of each <value expression> associated with the <field name> 'millisecond' is not between 0 (zero) and 999, or the value associated with 'microsecond', not between 0 (zero) and 999999 and the value for 'nanosecond' not between 0 (zero) and 999999999, or, if more than one of 'millisecond', 'microsecond', and 'nanosecond' is specified, any value exceeds 999, then an exception is raised: *data exception — invalid datetime function value (22G06)*.
- 6) If the values associated with other <field name>s do not conform to the range of values specified in [ISO 8601-1:2019](#), 4.3, “Time scale components and units”, then an exception is raised: *data exception — invalid datetime function value (22G06)*.
- 7) Let *DS* be the character string defined as follows.

Case:

- a) If <date function parameters> is specified using a <date string>, then *DS* is that <date string>.
- b) If <date function parameters> or <datetime function parameters> are specified using a <record constructor>, then:
 - i) Let *RVS* be the <record constructor> immediately contained in the <date function parameters> or the <datetime function parameters>.
 - ii) Let *YYS* be defined as follows.
 - 1) Let *YY* be the result of the <value expression> in the 'year' field of *RVS*.
 - 2) Let *YYC* be the result of:
`CAST(YY AS STRING)`
 - 3) If `CHAR_LENGTH(YYC) < 4`, then *YYS* is *YYC* prepended with 0's (zeroes) to a length of 4; otherwise, *YYS* is *YYC*.
 - iii) Let *DDS* be defined as follows.
 - 1) Let *DD* be defined as follows. If *RVS* does not specify the 'day' field, then *DD* is 1 (one); otherwise, *DD* is the result of the <value expression> in the 'day' field of *RVS*.
 - 2) Let *DDC* be the result of:
`CAST(DD AS STRING)`
 - 3) If `CHAR_LENGTH(DDC) < 2`, then *DDS* is *DDC* prepended with 0's (zeroes) to a length of 2; otherwise, *DDS* is *DDC*.
 - iv) Let *MMS* be defined as follows.
 - 1) Let *MM* be defined as follows. If *RVS* does not specify the 'month' field, then *MM* is 1 (one); otherwise, *MM* is the result of the <value expression> in the 'month' field of *RVS*.
 - 2) Let *MMC* be the result of:
`CAST(MM AS STRING)`
 - 3) If `CHAR_LENGTH(MMC) < 2`, then *MMS* is *MMC* prepended with 0's (zeroes) to a length of 2; otherwise, *MMS* is *MMC*.
 - v) *DS* is the result of:
`YYS || '-' || MMS || '-' || DDS`

without intervening whitespace.

c) Otherwise, *DS* is the zero-length character string.

8) Let *TS* be the character string defined as follows.

Case:

a) If <time function parameters> is specified using a <time string>, then *TS* is that <time string>.

b) If <time function parameters> or <datetime function parameters> are specified using a <record constructor>, then:

i) Let *RVS* be the <record constructor> immediately contained in the <time function parameters> or the <datetime function parameters>.

ii) Let *HRS* be defined as follows.

1) Let *HR* be the result of the <value expression> in the 'hour' field of *RVS*.

2) Let *HRC* be the result of:

`CAST(HR AS STRING)`

3) If `CHAR_LENGTH(HRC) < 2`, then *HRS* is *HRC* prepended with 0's (zeroes) to a length of 2; otherwise, *HRS* is *HRC*.

iii) Let *MINS* be defined as follows.

1) Let *MIN* be defined as follows. If *RVS* does not specify the 'minute' field, then *MIN* is 0 (zero); otherwise, *MIN* is the result of the <value expression> in the 'minute' field of *RVS*.

2) Let *MINC* be the result of:

`CAST(MIN AS STRING)`

3) If `CHAR_LENGTH(MINC) < 2`, then *MINS* is *MINC* prepended with 0's (zeroes) to a length of 2; otherwise, *MINS* is *MINC*.

iv) Let *SECS* be defined as follows.

1) Let *SEC* be defined as follows. If *RVS* does not specify the 'second' field, then *SEC* is 0 (zero); otherwise, *SEC* is the result of the <value expression> in the 'seconds' field of *RVS*.

2) Let *SECC* be the result of:

`CAST(SEC AS STRING)`

3) If `CHAR_LENGTH(SECC) < 2`, then *SECS* is *SECC* prepended with 0's (zeroes) to a length of 2; otherwise, *SECS* is *SECC*.

v) Let *SUBSECS* be defined as follows.

Case:

1) If *RVS* specifies the 'millisecond' field, then:

A) Let *MISEC* be the result of the <value expression> in the 'millisecond' field of *RVS*.

B) Let *MISECC* be the result of:

CAST(MISEC AS STRING)

- C) If CHAR_LENGTH(MISECC) < 3, then *SUBSECS* is *MISECC* prepended with 0's (zeroes) to a length of 3; otherwise, *SUBSECS* is *MISECC*.

- 2) If *RVS* specifies the 'microsecond' field, then:

- A) Let *MYSEC* be the result of the <value expression> in the 'microsecond' field of *RVS*.

- B) Let *MYSECC* be the result of:

CAST(MYSEC AS STRING)

- C) If CHAR_LENGTH(MYSECC) < 6, then *SUBSECS* is *MYSECC* prepended with 0's (zeroes) to a length of 6; otherwise, *SUBSECS* is *MYSECC*.

- 3) If *RVS* specifies the 'nanosecond' field, then:

- A) Let *NASEC* be the result of the <value expression> in the 'nanosecond' field of *RVS*.

- B) Let *NASECC* be the result of:

CAST(NASEC AS STRING)

- C) If CHAR_LENGTH(NASECC) < 9, then *SUBSECS* is *NASECC* prepended with 0's (zeroes) to a length of 9; otherwise, *SUBSECS* is *NASECC*.

- 4) Otherwise, *SUBSECS* is '000'.

- vi) Let *TZ* be defined as follows. If *RVS* contains the 'timezone' field, then *TZ* is the result of the <value expression> in the 'timezone' field of *RVS*; otherwise, *TZ* is the zero-length character string.

- vii) *TS* is the result of:

HRS || ':' || *MIN* || ':' || *SECS* || '.' || *SUBSECS* || *TZ*

without intervening whitespace.

- c) Otherwise, *TS* is the zero-length character string.

- 9) Let *DTS* be the character string defined as follows.

Case:

- a) If <datetime function parameters> is specified using a <datetime string>, then *DTS* is that <datetime string>.

- b) If <datetime function parameters> is specified using a <record constructor>, then:

- i) Let *RVS* be the <record constructor> immediately contained in the <datetime function parameters>.

- ii) *DTS* is the result of:

DS || 'T' || *TS*

without intervening whitespace.

- c) Otherwise, *DTS* is the zero-length character string.

10) If the <datetime value function> *DTVf* is specified, then

Case:

- a) If *DTVf* is *DATE()*, *ZONED_TIME()*, *ZONED_DATETIME()*, *TIME()*, or *DATETIME()*, then the result of *DTVf* respectively is the current date, zoned time, zoned datetime, local time, or local datetime from the current request timestamp.

For *ZONED_TIME()* and *ZONED_DATETIME()*, the result of *DTVf* has a time zone displacement that is equal to the current time zone displacement.

- b) Otherwise, *DTVf* is a <date function>, <time function>, <datetime function>, <localtime function>, or <localdatetime function> that specifies parameters and respectively the result of *DTVf* is the date, zoned time, zoned datetime, local time, or local datetime value associated with the respective representation of the parameters provided by *DS*, *TS*, *DTS*, *TS*, or *DTS*, as defined by ISO 8601-1:2019 and ISO 8601-2:2019.

Conformance Rules

- 1) Without Feature GV39, “Temporal types: date, local datetime and local time support”, conforming GQL language shall not contain a <datetime value function> that is a <localdatetime function>, a <date function>, or a <localtime function>.
- 2) Without Feature GV40, “Temporal types: zoned datetime and zoned time support”, conforming GQL language shall not contain a <datetime value function> that is a <datetime function> or a <time function>.

20.28 <duration value expression>

Function

Specify a duration value.

Format

```
<duration value expression> ::=
    <duration term>
    | <duration addition and subtraction>
    | <datetime subtraction>

<duration addition and subtraction> ::=
    <duration value expression 1> <plus sign> <duration term 1>
    | <duration value expression 1> <minus sign> <duration term 1>

<datetime subtraction> ::=
    DURATION_BETWEEN <left paren> <datetime subtraction parameters> <right paren>
    [ <temporal duration qualifier> ]

<datetime subtraction parameters> ::=
    <datetime value expression 1> <comma> <datetime value expression 2>

<duration term> ::=
    <duration factor>
    | <duration term 2> <asterisk> <factor>
    | <duration term 2> <solidus> <factor>
    | <term> <asterisk> <duration factor>

<duration factor> ::=
    [ <sign> ] <duration primary>

<duration primary> ::=
    <value expression primary>
    | <duration value function>

<duration value expression 1> ::=
    <duration value expression>

<duration term 1> ::=
    <duration term>

<duration term 2> ::=
    <duration term>

<datetime value expression 1> ::=
    <datetime value expression>

<datetime value expression 2> ::=
    <datetime value expression>
```

Syntax Rules

- 1) If the <duration value expression> immediately contains <duration addition and subtraction>, then:
 - a) The declared type of <duration value expression 1> and <duration term 1> shall be of the same duration unit group.

- b) The declared type of the <duration value expression> is the declared type of the <duration value expression 1>.
- 2) If the <duration value expression> immediately contains a <duration term>, then
Case:
 - a) If the <duration term> immediately contains a <duration factor>, then the declared type of the <duration value expression> is the declared type of the <duration factor>.
 - b) Otherwise, the declared type of the <duration value expression> is the declared type of the <duration term 2>.
- 3) If the <duration value expression> *DVE* immediately contains a <datetime subtraction> *DS*, then:
 - a) Let *DVE1* and *DVE2* be the <datetime value expression 1> and the <datetime value expression 2>, respectively, that are simply contained in *DS*.
 - b) The declared type of *DVE1* and the declared type of *DVE2* shall be essentially comparable value types.
 - c) Let *TDQ* be defined as follows. If *DS* simply contains a <temporal duration qualifier>, then *TDQ* is that <temporal duration qualifier>; otherwise, *TDQ* is DAY TO SECOND.
 - d) If *DVE1* is a zoned time type or a local time type and *DVE2* is a zoned time type or a local time type, then *TDQ* shall be DAY TO SECOND.
 - e) If *TDQ* is DAY TO SECOND, then the declared type of *DVE* shall be the day and time-based duration type; otherwise, the declared type of *DVE* shall be the year and month-based duration type.
- 4) If <duration term>, *DT* immediately contains <solidus>, then let *DT2* be the <duration term 2> immediately contained in *DT* and *F* be the <factor> immediately contained in *DT*. *DT* is effectively replaced by:

$$DT2 * (1 / F)$$

General Rules

- 1) If the result of a <value expression primary> simply contained in a <duration primary> is not a duration, then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 2) If the most specific types of <datetime value expression 1> and <datetime value expression 2> are not temporal instant types, then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 3) If <datetime value expression> is specified, then if the results of <datetime value expression> and <datetime primary> are not essentially comparable values, then an exception condition is raised: *data exception — values not comparable (22G04)*.
- 4) If <duration term> immediately contains <asterisk>, then let *V* be the result of the immediately contained <term> or <factor>. The result of the <duration term> is the duration specified by ISO 8601-2:2019, 14.3, "Multiplication", with *V* as the coefficient, and <duration term 2> or <duration factor> as 'durationA', where 'duration', 'coefficient', and 'durationA' are the terms used in ISO 8601-2:2019.
- 5) The result of a <duration value expression> *DVE* is defined as follows.
Case:

ISO/IEC 39075:2024(en)
20.28 <duration value expression>

- a) If the result of a <datetime value expression>, <duration value expression>, or <duration term> simply contained in *DVE* is the null value, then the result of *DVE* is the null value.
- b) If *DVE* immediately contains <duration term>, then the result of *DVE* is the result of <duration term>.
- c) If *DVE* immediately contains <duration addition and subtraction>, then
 Case:
 - i) If the two operands of *DVE* are of the same unit group, then the result of *DVE* is as specified by ISO 8601-2:2019, 14, “Date and time arithmetic”.
 - ii) Otherwise, the two operands of *DVE* are not of the same unit group and an exception condition is raised: *data exception — incompatible temporal instant unit groups* (22G14).
- d) If *DVE* immediately contains a <datetime subtraction> *DS*, then:
 - i) Let *DVE1* be the <datetime value expression 1> simply contained in *DS*.
 - ii) Let *DVE2* be the <datetime value expression 2> simply contained in *DS*.
 - iii) Let *MSP* be the implementation-defined (IL024) maximum value of fractional seconds precision for a temporal duration.
 - iv) Let the units *Y* be defined as follows. If *DVE1* and *DVE2* are datetimes or dates, then *Y* is Days; otherwise, *Y* is Seconds with a precision of *MSP*.
 - v) *DVE1* and *DVE2* are converted to integer scalars *DVE1IS* and *DVE2IS*, respectively, in units *Y* as displacements from some implementation-dependent (UV014) start datetime.
 - vi) The result of *DVE* is determined by:
 - 1) Effectively computing *RESULT* as *DVE2IS* minus *DVE1IS*, rounding or truncating if necessary.
 - 2) The result of *DVE* is *RESULT* converted to a duration of the temporal duration unit group specified by the <temporal duration qualifier>, rounding or truncating if necessary.
 - 3) The choice of whether to round or truncate is implementation-defined (IA005).
 If the required number of significant digits of the resulting duration exceeds *MSP*, then an exception condition is raised: *data exception — interval field overflow* (22O15).
 It is implementation-defined (IA026) whether the resulting duration reflects leap seconds.
- 6) The result of *DVE* is normalized, carrying over to the next most significant datetime field as needed, such that:
 - a) If the result of *DVE* is a day and time-based duration, then:
 - i) The absolute value of Seconds of is normalized to be < 60.
 - ii) The absolute value of Minutes is normalized to be < 60.
 - iii) The absolute value of Hours is normalized to be < 24.
 - b) Otherwise, the result of *DVE* is a year and month-based duration and the absolute value of Months is normalized to be < 12.

Conformance Rules

- 1) Without Feature GV41, “Temporal types: duration support”, conforming GQL language shall not contain a <duration value expression>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

20.29 <duration value function>

Function

Specify a function yielding a value of a temporal duration type.

Format

```
<duration value function> ::=  
    <duration function>  
    | <duration absolute value function>  
  
<duration function> ::=  
    DURATION <left paren> <duration function parameters> <right paren>  
  
<duration function parameters> ::=  
    <duration string>  
    | <record constructor>  
  
<duration absolute value function> ::=  
    ABS <left paren> <duration value expression> <right paren>
```

Syntax Rules

None.

General Rules

- 1) If <duration function parameters> immediately contains a <duration string> *DS*, then
Case:
 - a) If *DS* conforms to the Format of <iso8601 years and months> without a <separator> between the fields, then the declared type of the <duration function> is the year and month-based duration type.
 - b) If *DS* conforms to the Format of <iso8601 days and time> without a <separator> between the fields, then the declared type of the <duration function> is the day and time-based duration type.
 - c) Otherwise, an exception condition is raised: *data exception — invalid duration format (22G0H)*.
- 2) If <duration function parameters> immediately contains a <record constructor> *RVS*, then
Case:
 - a) If the declared type of *RVS* specifies the fields 'years' or 'months', then the declared type of the <duration function> is the year and month-based duration type.
 - b) If the declared type of *RVS* specifies the fields 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', or 'nanoseconds', then the declared type of the <duration function> is the day and time-based duration type.
 - c) Otherwise, an exception condition is raised: *data exception — invalid duration function field name (22G07)*.

ISO/IEC 39075:2024(en)
20.29 <duration value function>

- 3) If <duration function parameters> immediately contains a <record constructor>, then the record is transformed into an equivalent <duration string> as follows:
- a) If *RVS* specifies the fields 'years' or 'months', then:
- i) Let *YY* be defined as follows:
- 1) Let *YY* be defined as follows. If *RVS* does not specify the 'years' field, then *YY* is 0 (zero); otherwise, *YY* is the result of the <value expression> in the 'years' field of *RVS*.
 - 2) *YY* is the result of:
`CAST(YY AS STRING)`
- ii) Let *MMS* be defined as follows:
- 1) Let *MM* be defined as follows. If *RVS* does not specify the 'month' field, then *MM* is 0 (zero); otherwise, *MM* is the result of the <value expression> in the 'months' field of *RVS*.
 - 2) *MMS* is the result of:
`CAST(MM AS STRING)`
- iii) Let *DS* be the result of:
`'P' || YYS || 'Y' || MMS || 'M'`
- b) Otherwise,
- i) Let *DDS* be defined as follows:
- 1) Let *DD* be defined as follows. If *RVS* does not specify the 'days' field, then *DD* is 0 (zero); otherwise, *DD* is the result of the <value expression> in the 'days' field of *RVS*.
 - 2) *DDS* is the result of:
`CAST(DD AS STRING)`
- ii) Let *HRS* be defined as follows:
- 1) Let *HR* be defined as follows. If *RVS* does not specify the 'hours' field, then *HR* is 0 (zero); otherwise, *HR* is the result of the <value expression> in the 'hours' field of *RVS*.
 - 2) *HRS* is the result of:
`CAST(HR AS STRING)`
- iii) Let *MINS* be defined as follows:
- 1) Let *MIN* be defined as follows. If *RVS* does not specify the 'minutes' field, then *MIN* is 0 (zero); otherwise, *MIN* is the result of the <value expression> in the 'minutes' field of *RVS*.
 - 2) *MINS* is the result of:
`CAST(MIN AS STRING)`
- iv) Let *SECS* be defined as follows:

ISO/IEC 39075:2024(en)
20.29 <duration value function>

- 1) Let *SEC* be defined as follows. If *RVS* does not specify the 'seconds' field, then *SEC* is 0 (zero); otherwise, *SEC* is the result of the <value expression> in the 'seconds' field of *RVS*.
- 2) *SECS* is the result of:
`CAST(SEC AS STRING)`
- v) Let *SUBSECS* be defined as follows.
Case:
 - 1) If *RVS* specifies the 'milliseconds' field, then:
 - A) Let *MISEC* be the result of the <value expression> in the 'milliseconds' field of *RVS*.
 - B) Let *MISECC* be the result of:
`CAST(MISEC AS STRING)`
 - C) If `CHAR_LENGTH(MISECC) < 3`, then *SUBSECS* is *MISECC* prepended with 0's (zeroes) to a length of 3; otherwise, *SUBSECS* is *MISECC*.
 - 2) If *RVS* specifies the 'microseconds' field, then:
 - A) Let *MYSEC* be the result of the <value expression> in the 'microseconds' field of *RVS*.
 - B) Let *MYSECC* be the result of:
`CAST(MYSEC AS STRING)`
 - C) If `CHAR_LENGTH(MYSECC) < 6`, then *SUBSECS* is *MYSECC* prepended with 0's (zeroes) to a length of 6; otherwise, *SUBSECS* is *MYSECC*.
 - 3) If *RVS* specifies the 'nanoseconds' field, then:
 - A) Let *NASEC* be the result of the <value expression> in the 'nanoseconds' field of *RVS*.
 - B) Let *NASECC* be the result of:
`CAST(NASEC AS STRING)`
 - C) If `CHAR_LENGTH(NASECC) < 9`, then *SUBSECS* is *NASECC* prepended with 0's (zeroes) to a length of 9; otherwise, *SUBSECS* is *NASECC*.
 - 4) Otherwise, *SUBSECS* is '000'.
- vi) Let *SECSNSUBSECS* be the result of:
`SECS || '.' || SUBSECS || 'S'`
without intervening whitespace.
- vii) *DS* is the result of:
`'P' || DDS || 'DT' || HRS || 'H' || MINS || 'M' || SECSNSUBSECS`
without intervening whitespace.

ISO/IEC 39075:2024(en)
20.29 <duration value function>

- 4) The result of the <duration function> is the duration represented by the immediately contained <duration string>, as defined in [ISO 8601-1:2019](#), 5.5.2, “Duration” as extended by [ISO 8601-2:2019](#), 4.3, “Additional explicit forms” and [ISO 8601-2:2019](#), 4.4, “Numerical extensions”.
- 5) If <duration absolute value function> is specified, then let N be the result of the <duration 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

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

21 Lexical elements

21.1 Names and variables

Function

Specify names and variables.

Format

```

<authorization identifier> ::=
  <identifier>

<object name> ::=
  <identifier>

<object name or binding variable> ::=
  <regular identifier>

<directory name> ::=
  <identifier>

<schema name> ::=
  <identifier>

<graph name> ::=
  <regular identifier>
  | <delimited graph name>

<delimited graph name> ::=
  <delimited identifier>

<graph type name> ::=
  <identifier>

<node type name> ::=
  <identifier>

<edge type name> ::=
  <identifier>

<binding table name> ::=
  <regular identifier>
  | <delimited binding table name>

<delimited binding table name> ::=
  <delimited identifier>

<procedure name> ::=
  <identifier>

<label name> ::=
  <identifier>

<property name> ::=
  <identifier>

```

```
<field name> ::=  
  <identifier>  
  
<parameter name> ::=  
  <separated identifier>  
  
<graph pattern variable> ::=  
  <element variable>  
  | <path or subpath variable>  
  
<path or subpath variable> ::=  
  <path variable>  
  | <subpath variable>  
  
<element variable> ::=  
  <binding variable>  
  
<path variable> ::=  
  <binding variable>  
  
<subpath variable> ::=  
  <regular identifier>  
  
<binding variable> ::=  
  <regular identifier>
```

Syntax Rules

- 1) The <identifier>s that are valid <authorization identifier>s are implementation-defined (IV015).
- 2) The *name* specified by a <delimited identifier> or a <non-delimited identifier> *DIONDI* is the canonical name form of *DIONDI*.

NOTE 335 — See Syntax Rule 24) of Subclause 21.3, “<token>, <separator>, and <identifier>” for the definition of canonical name form.
- 3) A *system-generated name* is the canonical name form of a system-generated identifier.

General Rules

- 1) An <authorization identifier> identifies a principal and a set of privileges for that principal.
- 2) An <object name> identifies a GQL-object.
- 3) An <object name or binding variable> identifies a GQL-object or a binding variable.
- 4) A <directory name> identifies a GQL-directory.
- 5) A <schema name> identifies a GQL-schema.
- 6) A <graph name> identifies a graph.
- 7) A <delimited graph name> identifies a graph.
- 8) A <graph type name> identifies a graph type.
- 9) A <node type name> identifies a node type.
- 10) An <edge type name> identifies an edge type.
- 11) A <binding table name> identifies a binding table.
- 12) A <delimited binding table name> identifies a binding table.

- 13) A <procedure name> identifies a procedure.
- 14) A <label name> identifies a label.
- 15) A <property name> identifies a property of a GQL-object.
- 16) A <field name> identifies a field of a record, a field type of a record, or a column of a binding table.
- 17) A <parameter name> identifies a parameter.
- 18) A variable is a graph variable, a graph pattern variable, a binding table variable, a value variable, or a binding variable.
- 19) A <graph pattern variable> identifies a graph pattern variable. A graph pattern variable is an element variable, a path variable, or a subpath variable.
- 20) A <path or subpath variable> identifies a path variable or a subpath variable.
- 21) An <element variable> identifies an element variable.
- 22) A <path variable> identifies a path variable. Path variables are binding variables.
- 23) A <subpath variable> identifies a subpath variable.
- 24) A <binding variable> identifies a binding variable.

Conformance Rules

None.

21.2 <literal>

Function

Specify a value.

Format

```
<literal> ::=  
    <signed numeric literal>  
    | <general literal>  
  
<unsigned literal> ::=  
    <unsigned numeric literal>  
    | <general literal>  
  
<general literal> ::=  
    <boolean literal>  
    | <character string literal>  
    | <byte string literal>  
    | <temporal literal>  
    | <duration literal>  
    | <null literal>  
    | <list literal>  
    | <record literal>  
  
<boolean literal> ::=  
    TRUE | FALSE | UNKNOWN  
  
<character string literal> ::=  
    <single quoted character sequence>  
    | <double quoted character sequence>  
  
<single quoted character sequence> ::=  
    [ <no escape> ] <unbroken single quoted character sequence>  
  
<double quoted character sequence> ::=  
    [ <no escape> ] <unbroken double quoted character sequence>  
  
<accent quoted character sequence> ::=  
    [ <no escape> ] <unbroken accent quoted character sequence>  
  
<no escape> ::=  
    <commercial at>  
  
<unbroken single quoted character sequence> ::=  
    <quote> [ <single quoted character representation>... ] <quote>  
  
<unbroken double quoted character sequence> ::=  
    <double quote> [ <double quoted character representation>... ] <double quote>  
  
<unbroken accent quoted character sequence> ::=  
    <grave accent> [ <accent quoted character representation>... ] <grave accent>  
  
<single quoted character representation> ::=  
    <character representation>  
    | <double single quote>  
    !! See the Syntax Rules.  
  
<double quoted character representation> ::=  
    <character representation>
```

ISO/IEC 39075:2024(en)
21.2 <literal>

| <double double quote>
!! See the Syntax Rules.

<accent quoted character representation> ::=
 <character representation>
 | <double grave accent>
 !! See the Syntax Rules.

<character representation> ::=
 !! See the Syntax Rules.

<double single quote> ::=
 <quote> <quote>
 !! See the Syntax Rules.

<double double quote> ::=
 <double quote> <double quote>
 !! See the Syntax Rules.

<double grave accent> ::=
 <grave accent> <grave accent>
 !! See the Syntax Rules.

<string literal character> ::=
 !! See the Syntax Rules.

<escaped character> ::=
 <escaped reverse solidus>
 | <escaped quote>
 | <escaped double quote>
 | <escaped grave accent>
 | <escaped tab>
 | <escaped backspace>
 | <escaped newline>
 | <escaped carriage return>
 | <escaped form feed>
 | <unicode escape value>

<escaped reverse solidus> ::=
 <reverse solidus> <reverse solidus>

<escaped quote> ::=
 <reverse solidus> <quote>

<escaped double quote> ::=
 <reverse solidus> <double quote>

<escaped grave accent> ::=
 <reverse solidus> <grave accent>

<escaped tab> ::=
 <reverse solidus> t

<escaped backspace> ::=
 <reverse solidus> b

<escaped newline> ::=
 <reverse solidus> n

<escaped carriage return> ::=
 <reverse solidus> r

<escaped form feed> ::=
 <reverse solidus> f

ISO/IEC 39075:2024(en)
21.2 <literal>

```
<unicode escape value> ::=
    <unicode 4 digit escape value>
  | <unicode 6 digit escape value>

<unicode 4 digit escape value> ::=
    <reverse solidus> u <hex digit> <hex digit> <hex digit> <hex digit>

<unicode 6 digit escape value> ::=
    <reverse solidus> U <hex digit> <hex digit> <hex digit> <hex digit> <hex digit> <hex digit>

<byte string literal> ::=
    X <quote> [ <space>... ]
      [ { <hex digit> [ <space>... ] <hex digit> [ <space>... ] }... ]
      <quote>
      [ { <separator> <quote> [ <space>... ] [ { <hex digit> [ <space>... ]
        <hex digit> [ <space>... ] }... ] <quote> }... ]

<signed numeric literal> ::=
    [ <sign> ] <unsigned numeric literal>

<sign> ::=
    <plus sign>
  | <minus sign>

<unsigned numeric literal> ::=
    <exact numeric literal>
  | <approximate numeric literal>

<exact numeric literal> ::=
    <unsigned decimal in scientific notation> <exact number suffix>
  | <unsigned decimal in common notation> [ <exact number suffix> ]
  | <unsigned decimal integer> <exact number suffix>
  | <unsigned integer>

<exact number suffix> ::=
    M

<unsigned decimal in scientific notation> ::=
    <mantissa> E <exponent>
    !! See the Syntax Rules.

<mantissa> ::=
    <unsigned decimal in common notation>
  | <unsigned decimal integer>

<exponent> ::=
    <signed decimal integer>

<unsigned decimal in common notation> ::=
    <unsigned decimal integer> { <period> [ <unsigned decimal integer> ] }
  | <period> <unsigned decimal integer>

<unsigned integer> ::=
    <unsigned decimal integer>
  | <unsigned hexadecimal integer>
  | <unsigned octal integer>
  | <unsigned binary integer>

<signed decimal integer> ::=
    [ <sign> ] <unsigned decimal integer>

<unsigned decimal integer> ::=
    <digit> [ { [ <underscore> ] <digit> }... ]
```

ISO/IEC 39075:2024(en)
21.2 <literal>

<unsigned hexadecimal integer> ::=
 0x { [<underscore>] <hex digit> }...

<unsigned octal integer> ::=
 0o { [<underscore>] <octal digit> }...

<unsigned binary integer> ::=
 0b { [<underscore>] <binary digit> }...

<approximate numeric literal> ::=
 <unsigned decimal in scientific notation> [<approximate number suffix>]
 | <unsigned decimal in common notation> <approximate number suffix>
 | <unsigned decimal integer> <approximate number suffix>

<approximate number suffix> ::=
 F | D

<temporal literal> ::=
 <date literal>
 | <time literal>
 | <datetime literal>
 | <SQL-datetime literal>

<date literal> ::=
 DATE <date string>

<time literal> ::=
 TIME <time string>

<datetime literal> ::=
 { DATETIME | TIMESTAMP } <datetime string>

<date string> ::=
 <character string literal>

<time string> ::=
 <character string literal>

<datetime string> ::=
 <character string literal>

<time zone string> ::=
 <character string literal>

<SQL-datetime literal> ::=
 !! See the Syntax Rules.

<duration literal> ::=
 DURATION <duration string>
 | <SQL-interval literal>

<duration string> ::=
 <character string literal>

<iso8601 years and months> ::=
 P [<iso8601 years>] [<iso8601 months>]

<iso8601 years> ::=
 <iso8601 sint> Y

<iso8601 months> ::=
 <iso8601 sint> M

<iso8601 days> ::=
 <iso8601 sint> D

ISO/IEC 39075:2024(en)
21.2 <literal>

```
<iso8601 days and time> ::=  
  P [ <iso8601 days> ] T [ <iso8601 hours> ] [ <iso8601 minutes> ] [ <iso8601 seconds> ]  
  
<iso8601 hours> ::=  
  <iso8601 sint> H  
  
<iso8601 minutes> ::=  
  <iso8601 sint> M  
  
<iso8601 seconds> ::=  
  <iso8601 sint> [ <period> <iso8601 uint> ] S  
  
<iso8601 sint> ::=  
  [ <minus sign> ] <unsigned decimal integer>  
  
<iso8601 uint> ::=  
  <unsigned decimal integer>  
  
<SQL-interval literal> ::=  
  !! See the Syntax Rules.  
  
<null literal> ::=  
  NULL  
  
<list literal> ::=  
  <list value constructor by enumeration>  
  
<record literal> ::=  
  <record constructor>
```

Syntax Rules

- 1) In an <unsigned hexadecimal integer>, <unsigned octal integer>, or <unsigned binary integer>, there shall be no <separator> between the radix indicators “0x”, “0o”, “0b” and the first <hex digit>, <octal digit>, <binary digit>, or <underscore>.
- 2) An <unsigned decimal integer> that immediately contains <underscore>s is equivalent to the same <unsigned decimal integer> with every <underscore> removed.
- 3) An <unsigned hexadecimal integer> that immediately contains <underscore>s is equivalent to the same <unsigned hexadecimal integer> with every <underscore> removed.
- 4) An <unsigned octal integer> that immediately contains <underscore>s is equivalent to the same <unsigned octal integer> with every <underscore> removed.
- 5) An <unsigned binary integer> that immediately contains <underscore>s is equivalent to the same <unsigned binary integer> with every <underscore> removed.
- 6) An <unsigned hexadecimal integer> is equivalent to an <unsigned decimal integer> containing a series of <digit>s that represent the same integer number as the series of <hex digit>s.
- 7) An <unsigned octal integer> is equivalent to an <unsigned decimal integer> containing a series of <digit>s that represent the same integer number as the series of <octal digit>s.
- 8) An <unsigned binary integer> is equivalent to an <unsigned decimal integer> containing a series of <digit>s that represent the same integer number as the series of <binary digit>s.
- 9) In an <unsigned decimal in scientific notation>, the exponent indicator “E” may be represented either in upper-case (as “E”) or in lower-case (as “e”).
- 10) In an <approximate numeric literal>, the <approximate number suffix> “F” or “D” may be represented either in upper-case (as “F” or “D”) or in lower-case (as “f” or “d”).

- 11) There shall be no <separator> before the <approximate number suffix> in an <approximate numeric literal>.
- 12) If an <exact numeric literal> *ENL* is specified that immediately contains an <unsigned decimal in scientific notation> *UDSN* then:
 - a) Let *M* and *E* be the specified value of the <mantissa> and the <exponent>, respectively, that are simply contained in *UDSN*.
 - b) Let *UDCN* be an <unsigned decimal in common notation> whose specified value is $M \times 10^E$.
 - c) *ENL* is effectively replaced by:
UDCN
- 13) If an <approximate numeric literal> *ANL* is specified that does not immediately contain an <unsigned decimal in scientific notation>, then:
 - a) Let *DIGITS_BEFORE* and *DIGITS_AFTER* be the sequence of all <digit>s specified by *ANL* before, and, respectively, after the decimal point but with all leading and, respectively, trailing zero digits removed.
 - b) Let *M* be defined as the concatenation of *DIGITS_BEFORE* and *DIGITS_AFTER*.
 - c) Let *E* be an <unsigned integer> whose specified value is the negated length of *DIGITS_AFTER*.
 - d) *ANL* is effectively replaced by:
M E E
- 14) The maximum number of <digit>s immediately contained in an <unsigned integer> is implementation-defined (IL010) but shall not be less than 9.
- 15) An <exact numeric literal> that is an <unsigned decimal integer> has an implicit <period> following the last <digit>.
- 16) In an <exact numeric literal> the <exact number suffix> “M” may be represented either in upper-case (as “M”) or in lower-case (as “m”).
- 17) There shall be no <separator> before the <exact number suffix> in an <exact numeric literal>.
- 18) The declared type of an <exact numeric literal> *ENL* is an exact numeric type whose scale is the number of <digit>s to the right of the <period>. There shall be an exact numeric type capable of representing the value of *ENL* exactly.
- 19) The declared type of an <approximate numeric literal> *ANL* is defined as follows.
Case:
 - a) If *ANL* simply contains the <approximate number suffix> “F” or “f”, then the declared type of *ANL* is the regular approximate numeric type.
 - b) If *ANL* simply contains the <approximate number suffix> “D” or “d”, then the declared type of *ANL* is the double approximate numeric type.
 - c) Otherwise, the declared type of *ANL* is an implementation-defined (ID079) approximate numeric type.
- 20) The value of an <approximate numeric literal> *ANL* shall not be greater than the maximum value or less than the minimum value that can be represented by the declared type of *ANL*.
- 21) The declared type of a <boolean literal> is the Boolean type.

ISO/IEC 39075:2024(en)
21.2 <literal>

- 22) The <character string literal> specifies the character string specified by the immediately contained <single quoted character sequence> or the immediately contained <double quoted character sequence>.
- 23) The <single quoted character sequence> specifies the character string specified by the immediately contained <unbroken single quoted character sequence>.
- 24) The <double quoted character sequence> specifies the character string specified by the immediately contained <unbroken double quoted character sequence>.
- 25) The <accent quoted character sequence> specifies the character string specified by the immediately contained <unbroken accent quoted character sequence>.
- 26) The <unbroken single quoted character sequence> specifies the character string comprising the sequence of characters defined by the immediately contained <single quoted character representation>s.
- 27) The <unbroken double quoted character sequence> specifies the character string comprising the sequence of characters defined by the immediately contained <double quoted character representation>s.
- 28) The <unbroken accent quoted character sequence> specifies the character string comprising the sequence of characters defined by the immediately contained <accent quoted character representation>s.
- 29) There shall be no <separator> between <no escape> and an <unbroken single quoted character sequence>, <unbroken double quoted character sequence>, or <unbroken accent quoted character sequence>.
- 30) For every BNF non-terminal instance *NT* simply contained in a <single quoted character sequence>, <double quoted character sequence>, or <accent quoted character sequence> *CS*, if *CS* immediately contains the <no escape>, then character escaping is said to be disabled in *NT*; otherwise, character escaping is said to be enabled in *NT*.
- 31) If <single quoted character representation> *CR* is specified, then:
 - a) If character escaping is enabled in *CR*, then *CR* shall not be a <quote> or a <reverse solidus> unless either of these occurs as part of an <escaped character>.
 - b) Otherwise, character escaping is disabled in *CR* and *CR* shall not be a <quote>.
- 32) If <double quoted character representation> *CR* is specified, then:
 - a) If character escaping is enabled in *CR*, then *CR* shall not be a <double quote> or a <reverse solidus> unless either of these occurs as part of an <escaped character>.
 - b) Otherwise, character escaping is disabled in *CR* and *CR* shall not be a <double quote>.
- 33) If <accent quoted character representation> *CR* is specified, then:
 - a) If character escaping is enabled in *CR*, then *CR* shall not be a <grave accent> or a <reverse solidus> unless either of these occurs as part of an <escaped character>.
 - b) Otherwise, character escaping is disabled in *CR* and *CR* shall not be a <grave accent>.
- 34) In a <single quoted character representation>, each <double single quote> represents a single <quote> character.
- 35) In a <double quoted character representation>, each <double double quote> represents a single <double quote> character.

- 36) In an <accent quoted character representation>, each <double grave accent> represents a single <grave accent> character.
- 37) In a <double single quote>, there shall be no <separator> after the leading <quote>.
- 38) In a <double double quote>, there shall be no <separator> after the leading <double quote>.
- 39) In a <double grave accent>, there shall be no <separator> after the leading <grave accent>.
- 40) If character escaping is enabled in a <character representation> *CR*, then *CR* shall conform to the Format and Syntax Rules of a <string literal character> or an <escaped character>; otherwise, character escaping is disabled in *CR* and *CR* shall conform to the Format and Syntax Rules of <string literal character> only.
- 41) If the <character representation> *CR* is specified, then:
- a) Case:
 - i) If character escaping is enabled in *CR*, then any character in *CR* except for those occurring as part of an <escaped character> is considered a <string literal character>.
 - ii) Otherwise, character escaping is disabled in *CR* and any character in *CR* is considered a <string literal character>.
 - b) For every <string literal character> *CH* contained in *CR*:
 - i) The character *CH* is a character in the GQL source text character repertoire.
NOTE 336 — The character repertoire of GQL source text is specified in Subclause 4.7.1, “General description of GQL-requests and GQL-programs”.
 - ii) If *CH* is not <whitespace>, then it shall not be in the Unicode General Category Class “Cc”.
NOTE 337 — In particular, this excludes BACKSPACE (U+0008).
 - iii) Whether *CH* shall not be a <bidirectional control character> is implementation-defined (IA019).
NOTE 338 — This mitigates CVE-2021-42574.
 - iv) *CH* shall not be in the Unicode General Category Class “Cn”.
NOTE 339 — The Unicode General Category classes “Cc” and “Cn”, are assigned to Unicode characters that are, respectively, Control codes (other) and Not Assigned codes.
NOTE 340 — Despite these restrictions, any character can still be expressed in a <character representation> through the use of a suitable <escaped character>.
- 42) In an <escaped character>, there shall be no <separator> after the leading <reverse solidus>.
- 43) In an <escaped character>, each <escaped reverse solidus> represents a <reverse solidus> character.
- 44) In an <escaped character>, each <escaped quote> represents a <quote> character.
- 45) In an <escaped character>, each <escaped double quote> represents a <double quote> character.
- 46) In an <escaped character>, each <escaped grave accent> represents a <grave accent> character.
- 47) In an <escaped character>, each <escaped tab> represents the Unicode character identified by the code point U+0009.
- 48) In an <escaped character>, each <escaped backspace> represents the Unicode character identified by the code point U+0008.

ISO/IEC 39075:2024(en)
21.2 <literal>

- 49) In an <escaped character>, each <escaped newline> represents the Unicode character identified by the code point U+000A.
- 50) In an <escaped character>, each <escaped carriage return> represents the Unicode character identified by the code point U+000D.
- 51) In an <escaped character>, each <escaped form feed> represents the Unicode character identified by the code point U+000C.
- 52) In an <escaped character>, each <unicode escape value> represents the Unicode character identified by the code point.
- 53) The declared type of a <character string literal> is character string.
- 54) In a <byte string literal>, the sequence
- ```
<quote> [<space>...] { <hex digit> [<space>...]
<hex digit> [<space>...] }... <quote>
```
- is equivalent to the sequence
- ```
<quote> { <hex digit> <hex digit> }... <quote>
```
- NOTE 341 — The <hex digit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <byte string literal>.
- 55) In a <byte string literal>, the sequence
- ```
<quote> { <hex digit> <hex digit> }... <quote> <separator>
<quote> { <hex digit> <hex digit> }... <quote>
```
- is equivalent to the sequence
- ```
<quote> { <hex digit> <hex digit> }... { <hex digit> <hex digit> }... <quote>
```
- NOTE 342 — The <hex digit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <byte string literal>.
- 56) In a <byte string literal>, the introductory 'X' may be represented either in upper-case (as 'X') or in lower-case (as 'x').
- 57) In a <byte string literal>, a <separator> shall contain a <newline>.
- 58) The declared type of a <byte string literal> is a byte string type. Each <hex digit> 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 <hex digit>s a, b, c, d, e, and f have respectively the same values as the <hex digit>s A, B, C, D, E, and F.
- 59) The character string specified by the <character string literal> immediately contained in <date string> shall conform to the representation specified in [ISO 8601-1:2019](#), 5.2, “Date”, as extended by [ISO 8601-2:2019](#), 4.3, “Additional explicit forms”, and [ISO 8601-2:2019](#), 4.4, “Numerical extensions”.
- 60) The declared type of <date literal> is the date type.
- 61) The character string specified by the <character string literal> immediately contained in <time zone string> shall conform to the representation specified in [ISO 8601-1:2019](#), 4.3.13, “Time shift”, as extended by [ISO 8601-2:2019](#), 4.3, “Additional explicit forms”, and [ISO 8601-2:2019](#), 4.4, “Numerical extensions”.
- 62) The character string specified by the <character string literal> immediately contained in <time string> shall conform to the representation specified in [ISO 8601-1:2019](#), 5.3, “Time of day”, as

ISO/IEC 39075:2024(en)
21.2 <literal>

extended by ISO 8601-2:2019, 4.3, “Additional explicit forms”, and ISO 8601-2:2019, 4.4, “Numerical extensions”.

- 63) If the <time string> does not contain a representation of a time zone displacement, then the declared type of <time literal> is the local time type; otherwise, the declared type of <time literal> is the zoned time type.
- 64) The character string specified by the <character string literal> immediately contained in <datetime string> shall conform to the representation specified in ISO 8601-1:2019, 5.4, “Date and time of day”, as extended by ISO 8601-2:2019, 4.3, “Additional explicit forms”, and ISO 8601-2:2019, 4.4, “Numerical extensions”.
- 65) If the <datetime string> does not contain a representation of a time zone displacement, then the declared type of <datetime literal> is the local datetime type, otherwise, the declared type of <datetime literal> is the zoned datetime type.
- 66) The declared type of <duration literal> is the duration type.
- 67) The nullable declared type *DT* of a <null literal> *NL* is defined as follows.
- Case:
- a) If *DT* can be determined by the context in which *NL* appears, then *NL* is effectively replaced by *CAST(NL AS DT)*.
- NOTE 343 — In every such context, *NL* is uniquely associated with some expression or site of (a nullable) declared type *DT*, which thereby becomes the declared type of *NL*.
- b) Otherwise, it is implementation-defined (IA014) whether an exception condition is raised: *syntax error or access rule violation — invalid syntax (42001)*. If an exception condition is not raised, then *DT* is most specific static value type of the null value.
- NOTE 344 — See Subclause 4.12.6, “Most specific static value type and static base type”, for the definition of the most specific static value type of the null value.
- 68) The character string specified by the <character string literal> immediately contained in <duration string> shall conform to one of the following:
- a) The Format of <iso8601 years and months>, including one or more of <iso8601 years> and <iso8601 months>, without a <separator> between the fields.
- b) The Format of <iso8601 days and time>, including one or more of <iso8601 days>, <iso8601 hours>, <iso8601 minutes>, and <iso8601 seconds>, without a <separator> between the fields.
- c) The representation specified in ISO 8601-1:2019, 5.5.2, “Duration”, as extended by ISO 8601-2:2019, 4.3, “Additional explicit forms”, and ISO 8601-2:2019, 4.4, “Numerical extensions”.
- 69) <SQL-datetime literal> shall conform to the Format and Syntax Rules of <datetime literal> in ISO/IEC 9075-2:2023.
- 70) <SQL-interval literal> shall conform to the Format and Syntax Rules of <interval literal> in ISO/IEC 9075-2:2023.
- 71) Case:
- a) If <duration string> is an <iso8601 years and months> or an <SQL-interval literal> that is a SQL <year-month literal>, then the declared type of the <duration literal> is the year and month-based duration type.

- b) If <duration string> is an <iso8601 days and time> or an <SQL-interval literal> that is a SQL <day-time literal>, then the declared type of the <duration literal> is the day and time-based duration type.
- c) Otherwise, an exception condition is raised: *data exception — invalid duration format (22G0H)*.

72) If <SQL-interval literal> is specified, then:

- a) If <SQL-interval literal> contains a <years value> *y*, then let *Y* be 'Yy'; otherwise, let *Y* be the zero-length character string.
- b) If <SQL-interval literal> contains a <months value> *m*, then let *M* be 'Mm'; otherwise, let *M* be the zero-length character string.
- c) If <SQL-interval literal> contains a <days value> *d*, then let *D* be 'Dd'; otherwise, let *D* be the zero-length character string.
- d) If <SQL-interval literal> contains an <hours value> *h*, then let *H* be 'Hh'; otherwise, let *H* be the zero-length character string.
- e) If <SQL-interval literal> contains a <minutes value> *mn*, then let *MN* be 'Mmn'; otherwise, let *MN* be the zero-length character string.
- f) If <SQL-interval literal> contains a <seconds value> *s*, then let *S* be 'Ss'; otherwise, let *S* be the zero-length character string.
- g) If <SQL-interval literal> contains an <hours value>, <minutes value>, or <seconds value>, then let *T* be 'T'; otherwise, let *T* be the zero-length character string.
- h) If <SQL-interval literal> contains at exactly one <sign> that is a <minus sign>, then let *SN* be '-'; otherwise, let *SN* be the zero-length character string.
- i) <SQL-interval literal> is equivalent to the <duration literal>:

DURATION 'SNPYMDTHMNS'

- 73) Every <value expression> contained in a <list element> simply contained in the <list value constructor by enumeration> contained in a <list literal> shall conform to the Format and Syntax Rules of <literal>.
- 74) The declared type of <list literal> is the declared type of the immediately contained <list value constructor by enumeration>.
- 75) Every <value expression> contained in a <field> simply contained in the <record constructor> contained in a <record literal> shall conform to the Format and Syntax Rules of <literal>.
- 76) The value specified by a <literal> is the value of that <literal> that is determined by a hypothetical application of its General Rules.

NOTE 345 — The values of certain <literal>s are specified by the General Rules of this Subclause for use in the evaluation of <value expression>s. However, such values are essentially constants and thus can be determined during the application of Syntax Rules.

General Rules

- 1) The value specified by a <literal> is the value represented by that <literal>.
- 2) Except when it is contained in an <exact numeric literal>, the value of an <unsigned integer> is the exact numeric value determined by application of the normal mathematical interpretation of positional decimal notation to the series of <digit>s that constitutes the <unsigned integer>.

ISO/IEC 39075:2024(en)
21.2 <literal>

- 3) Except when it is contained in an <exact numeric literal>, the value of an <unsigned decimal integer> is the exact numeric value determined by application of the normal mathematical interpretation of positional decimal notation to the source characters that constitute the <unsigned decimal integer>.
- 4) The value of an <exact numeric literal> is the exact numeric value determined by application of the normal mathematical interpretation of positional decimal notation to the source characters that constitute the <exact numeric literal>.
- 5) Let *ANL* be an <approximate numeric literal>. Let *ANDT* be the declared type of *ANL*. Let *ANV* be the product of the exact numeric value represented by the <mantissa> of *ANL* and the number obtained by raising the number 10 to the power of the exact numeric value represented by the <exponent> of *ANL*. If *ANV* is a value of *ANDT*, then the value of *ANL* is *ANV*; otherwise, the value of *ANL* is a value of *ANDT* obtained from *ANV* by rounding or truncation. The choice of whether to round or truncate is implementation-defined (IA005).
- 6) The <sign> in a <signed numeric 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 if monadic plus is specified, then the literal is positive. If monadic minus is specified in a <signed numeric literal>, then the literal is negative.
- 7) 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.
- 8) The value of a <character string literal> is the character string that it specifies.
- 9) The value of a <byte string literal> is the byte string comprising sequence of bits defined by the <hex digit>s that it contains. Each <hex digit> 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 <hex digit>s a, b, c, d, e, and f have respectively the same values as the <hex digit>s A, B, C, D, E, and F.
- 10) If <date string> contains a representation with reduced precision, then the lowest valid value for each of the omitted lower order time scale components is implicit.
- 11) The value of a <date literal> is a calendar date in the Gregorian calendar.
- 12) If <time string> contains a representation with reduced precision, then value for each of the omitted lower order time scale components is 0 (zero).
- 13) The value of a <time literal> is a time of day. If the <time string> contained a representation of a time shift, then the time shift information is preserved.
- 14) If <datetime string> contains a representation with reduced precision, then value for each of the omitted lower order time scale components is 0 (zero).
- 15) The value of a <datetime literal> is a time. If the <datetime string> contained a representation of a time shift, then the time shift information is preserved.
- 16) The value of a <duration literal> is a duration or a negative duration.
NOTE 346 — See ISO 8601-2:2019, 4.4.1.9 “Duration”, for the definition of a negative duration.
- 17) The value of a <null literal> is the null value.
- 18) The value of <list literal> is the value of the immediately contained <list value constructor by enumeration>.
- 19) The value of <record literal> is the value of the immediately contained <record constructor>.

Conformance Rules

- 1) Without Feature GL01, “Hexadecimal literals”, conforming GQL language shall not contain an <unsigned hexadecimal integer>.
- 2) Without Feature GL02, “Octal literals”, conforming GQL language shall not contain an <unsigned octal integer>.
- 3) Without Feature GL03, “Binary literals”, conforming GQL language shall not contain an <unsigned binary integer>.
- 4) Without Feature GL11, “Opt-out character escaping”, conforming GQL language shall not contain <no escape>.
- 5) Without Feature GL07, “Approximate number in common notation or as decimal integer with suffix”, conforming GQL language shall not contain an <approximate numeric literal> that is an <unsigned decimal in common notation> followed by an <approximate number suffix> or an <unsigned decimal integer> followed by an <approximate number suffix>.
- 6) Without Feature GL08, “Approximate number in scientific notation with suffix”, conforming GQL language shall not contain an <approximate numeric literal> that is an <unsigned decimal in scientific notation> followed by an <approximate number suffix>.
- 7) Without Feature GL09, “Optional float number suffix”, conforming GQL language shall not contain an <approximate numeric literal> that is an <unsigned decimal in scientific notation> followed by an <approximate number suffix>.
- 8) Without Feature GL10, “Optional double number suffix”, conforming GQL language shall not contain an <approximate numeric literal> that simply contains an <approximate number suffix> that is “d” or “D”.
- 9) Without Feature GL04, “Exact number in common notation without suffix”, conforming GQL language shall not contain an <exact numeric literal> that is an <unsigned decimal in common notation>.
- 10) Without Feature GL05, “Exact number in common notation or as decimal integer with suffix”, conforming GQL language shall not contain an <exact numeric literal> that is an <unsigned decimal in common notation> followed by an <exact number suffix> or an <unsigned decimal integer> followed by an <exact number suffix>.
- 11) Without Feature GL06, “Exact number in scientific notation with suffix”, conforming GQL language shall not contain an <exact numeric literal> that is an <unsigned decimal in scientific notation> followed by an <exact number suffix>.
- 12) Without Feature GV39, “Temporal types: date, local datetime and local time support”, conforming GQL language shall not contain a <temporal literal>.
- 13) Without Feature GV40, “Temporal types: zoned datetime and zoned time support”, conforming GQL language shall not contain a <temporal literal> that specifies a time zone displacement.
- 14) Without Feature GV41, “Temporal types: duration support”, conforming GQL language shall not contain a <duration literal>.
- 15) Without Feature GL12, “SQL datetime and interval formats”, conforming GQL language shall not contain an <SQL-datetime literal> or an <SQL-interval literal>.
- 16) Without Feature GV50, “List value types”, conforming GQL language shall not contain a <list literal>.

21.3 <token>, <separator>, and <identifier>

Function

Specify lexical units (tokens, separators, and identifiers) that participate in the GQL language.

Format

```
<token> ::=
    <non-delimiter token>
  | <delimiter token>

<non-delimiter token> ::=
    <regular identifier>
  | <substituted parameter reference>
  | <general parameter reference>
  | <keyword>
  | <unsigned numeric literal>
  | <byte string literal>
  | <multiset alternation operator>

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

<separated identifier> ::=
    <extended identifier>
  | <delimited identifier>

<non-delimited identifier> ::=
    <regular identifier>
  | <extended identifier>

<regular identifier> ::=
    <identifier start> [ <identifier extend>... ]

<extended identifier> ::=
    <identifier extend>...

<delimited identifier> ::=
    <double quoted character sequence>
  | <accent quoted character sequence>

<identifier start> ::=
    !! See the Syntax Rules.

<identifier extend> ::=
    !! See the Syntax Rules.

<substituted parameter reference> ::=
    <double dollar sign> <parameter name>

<general parameter reference> ::=
    <dollar sign> <parameter name>

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

<reserved word> ::=
    <pre-reserved word>
```

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

| ABS | ACOS | ALL | ALL_DIFFERENT | AND | ANY | ARRAY | AS | ASC | ASCENDING | ASIN
| AT | ATAN | AVG

| BIG | BIGINT | BINARY | BOOL | BOOLEAN | BOTH | BTRIM | BY | BYTE_LENGTH | BYTES

| CALL | CARDINALITY | CASE | CAST | CEIL | CEILING | CHAR | CHAR_LENGTH | CHARACTER_LENGTH
| CHARACTERISTICS | CLOSE | COALESCE | COLLECT_LIST | COMMIT | COPY | COS | COSH | COT
| COUNT | CREATE | CURRENT_DATE | CURRENT_GRAPH | CURRENT_PROPERTY_GRAPH
| CURRENT_SCHEMA | CURRENT_TIME | CURRENT_TIMESTAMP

| DATE | DATETIME | DAY | DEC | DECIMAL | DEGREES | DELETE | DESC | DESCENDING | DETACH
| DISTINCT | DOUBLE | DROP | DURATION | DURATION_BETWEEN

| ELEMENT_ID | ELSE | END | EXCEPT | EXISTS | EXP

| FALSE | FILTER | FINISH | FLOAT | FLOAT16 | FLOAT32 | FLOAT64 | FLOAT128 | FLOAT256
| FLOOR | FOR | FROM

| GROUP

| HAVING | HOME_GRAPH | HOME_PROPERTY_GRAPH | HOME_SCHEMA | HOUR

| IF | IMPLIES | IN | INSERT | INT | INTEGER | INT8 | INTEGER8 | INT16 | INTEGER16 |
INT32
| INTEGER32 | INT64 | INTEGER64 | INT128 | INTEGER128 | INT256 | INTEGER256 | INTERSECT
| INTERVAL | IS

| LEADING | LEFT | LET | LIKE | LIMIT | LIST | LN | LOCAL | LOCAL_DATETIME | LOCAL_TIME
| LOCAL_TIMESTAMP | LOG | LOG10 | LOWER | LTRIM

| MATCH | MAX | MIN | MINUTE | MOD | MONTH

| NEXT | NODETACH | NORMALIZE | NOT | NOTHING | NULL | NULLS | NULLIF

| OCTET_LENGTH | OF | OFFSET | OPTIONAL | OR | ORDER | OTHERWISE

| PARAMETER | PARAMETERS | PATH | PATH_LENGTH | PATHS | PERCENTILE_CONT | PERCENTILE_DISC
| POWER | PRECISION | PROPERTY_EXISTS

| RADIANS | REAL | RECORD | REMOVE | REPLACE | RESET | RETURN | RIGHT | ROLLBACK | RTRIM

| SAME | SCHEMA | SECOND | SELECT | SESSION | SESSION_USER | SET | SIGNED | SIN | SINH
| SIZE | SKIP | SMALL | SMALLINT | SQRT | START | STDDEV_POP | STDDEV_SAMP | STRING
| SUM

| TAN | TANH | THEN | TIME | TIMESTAMP | TRAILING | TRIM | TRUE | TYPED

| UBIGINT | UINT | UINT8 | UINT16 | UINT32 | UINT64 | UINT128 | UINT256 | UNION | UNKNOWN
| UNSIGNED | UPPER | USE | USMALLINT

| VALUE | VARBINARY | VARCHAR | VARIABLE

| WHEN | WHERE | WITH

| XOR

| YEAR | YIELD

| ZONED | ZONED_DATETIME | ZONED_TIME

<pre-reserved word> ::=

ABSTRACT | AGGREGATE | AGGREGATES | ALTER

| CATALOG | CLEAR | CLONE | CONSTRAINT | CURRENT_ROLE | CURRENT_USER

| DATA | DIRECTORY | DRYRUN

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

| EXACT | EXISTING
| FUNCTION
| GQLSTATUS | GRANT
| INSTANT | INFINITY
| NUMBER | NUMERIC
| ON | OPEN
| PARTITION | PROCEDURE | PRODUCT | PROJECT
| QUERY
| RECORDS | REFERENCE | RENAME | REVOKE
| SUBSTRING | SYSTEM_USER
| TEMPORAL
| UNIQUE | UNIT
| VALUES
| WHITESPACE
<non-reserved word> ::=
 ACYCLIC
| BINDING | BINDINGS
| CONNECTING
| DESTINATION | DIFFERENT | DIRECTED
| EDGE | EDGES | ELEMENT | ELEMENTS
| FIRST
| GRAPH | GROUPS
| KEEP
| LABEL | LABELED | LABELS | LAST
| NFC | NFD | NFKC | NFKD | NO | NODE | NORMALIZED
| ONLY | ORDINALITY
| PROPERTY
| READ | RELATIONSHIP | RELATIONSHIPS | REPEATABLE
| SHORTEST | SIMPLE | SOURCE
| TABLE | TEMP | TO | TRAIL | TRANSACTION | TYPE
| UNDIRECTED
| VERTEX
| WALK | WITHOUT | WRITE
| ZONE

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

<multiset alternation operator> ::=
|+| !! <U+007C, U+002B, U+007C>

<delimiter token> ::=
 <GQL special character>
| <bracket right arrow>
| <bracket tilde right arrow>
| <character string literal>
| <concatenation operator>
| <date string>
| <datetime string>
| <delimited identifier>
| <double colon>
| <double period>
| <duration string>
| <greater than operator>
| <greater than or equals operator>
| <left arrow>
| <left arrow bracket>
| <left arrow tilde>
| <left arrow tilde bracket>
| <left minus right>
| <left minus slash>
| <left tilde slash>
| <less than operator>
| <less than or equals operator>
| <minus left bracket>
| <minus slash>
| <not equals operator>
| <right arrow>
| <right bracket minus>
| <right bracket tilde>
| <right double arrow>
| <slash minus>
| <slash minus right>
| <slash tilde>
| <slash tilde right>
| <tilde left bracket>
| <tilde right arrow>
| <tilde slash>
| <time string>

<bracket right arrow> ::=
]-> !! <U+005D, U+002D, U+003E>

<bracket tilde right arrow> ::=
]~> !! <U+005D, U+007E, U+003E>

<concatenation operator> ::=
|| !! <U+007C, U+007C>

<double colon> ::=
:: !! <U+003A, U+003A>

<double dollar sign> ::=
\$\$!! <U+0024, U+0024>

<double minus sign> ::=
-- !! <U+002D, U+002D>

<double period> ::=
.. !! <U+002E, U+002E>

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

<greater than operator> ::=
 <right angle bracket>

<greater than or equals operator> ::=
 >= !! <U+003E, U+003D>

<left arrow> ::=
 <- !! <U+003C, U+002D>

<left arrow tilde> ::=
 <~ !! <U+003C, U+007E>

<left arrow bracket> ::=
 <-[!! <U+003C, U+002D, U+005B>

<left arrow tilde bracket> ::=
 <~[!! <U+003C, U+007E, U+005B>

<left minus right> ::=
 <-> !! <U+003C, U+002D, U+003E>

<left minus slash> ::=
 <- / !! <U+003C, U+002D, U+002F>

<left tilde slash> ::=
 <~/ !! <U+003C, U+007E, U+002F>

<less than operator> ::=
 <left angle bracket>

<less than or equals operator> ::=
 <= !! <U+003C, U+003D>

<minus left bracket> ::=
 -[!! <U+002D, U+005B>

<minus slash> ::=
 -/ !! <U+002D, U+002F>

<not equals operator> ::=
 <> !! <U+003C, U+003E>

<right arrow> ::=
 -> !! <U+002D, U+003E>

<right bracket minus> ::=
]- !! <U+005D, U+002D>

<right bracket tilde> ::=
]~ !! <U+005D, U+007E>

<right double arrow> ::=
 => !! <U+003D, U+003E>

<slash minus> ::=
 /- !! <U+002F, U+002D>

<slash minus right> ::=
 /-> !! <U+002F, U+002D, U+003E>

<slash tilde> ::=
 /~ !! <U+002F, U+007E>

<slash tilde right> ::=
 /~> !! <U+002F, U+007E, U+003E>

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

<tilde left bracket> ::=
~[!! <U+007E, U+005B>

<tilde right arrow> ::=
~> !! <U+007E, U+003E>

<tilde slash> ::=
~/ !! <U+007E, U+002F>

<double solidus> ::=
// !! <U+002F, U+002F>

<separator> ::=
{ <comment> | <whitespace> }...

<whitespace> ::=
!! See the Syntax Rules.

<truncating whitespace> ::=
!! See the Syntax Rules.

<bidirectional control character> ::=
!! See the Syntax Rules.

<comment> ::=
 <simple comment>
 | <bracketed comment>

<simple comment> ::=
 <simple comment introducer> [<simple comment character>...] <newline>

<simple comment introducer> ::=
 <double solidus>
 | <double minus sign>

<simple comment character> ::=
 !! See the Syntax Rules.

<bracketed comment> ::=
 <bracketed comment introducer>
 <bracketed comment contents>
 <bracketed comment terminator>

<bracketed comment introducer> ::=
 /* !! <U+002F, U+002A>

<bracketed comment terminator> ::=
 */ !! <U+002A, U+002F>

<bracketed comment contents> ::=
 !! See the Syntax Rules.

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

<edge synonym> ::=
 EDGE | RELATIONSHIP

<edges synonym> ::=
 EDGES | RELATIONSHIPS

<node synonym> ::=
 NODE | VERTEX

<implies> ::=

<right double arrow> | IMPLIES

Syntax Rules

- 1) An <identifier start> shall be <underscore> or any character with the Unicode property `XID_Start`, optionally modified by an implementation-defined (IE003) profile, in accordance with UAX31-D1 Default Identifier Syntax in [Unicode Standard Annex #31](#).

NOTE 347 — The characters in `ID_Start` are those in the Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, and “Nl” together with those with the Unicode property `Other_ID_Start` but none of which have the Unicode properties `Pattern_Syntax` or `Pattern_White_Space`. The characters in `XID_Start` are derived from those in `ID_Start` to ensure closure under NFKC normalization.

- 2) An <identifier extend> shall be any character with the Unicode property `XID_Continue`, optionally modified by an implementation-defined (IE003) profile, in accordance with UAX31-D1 Default Identifier Syntax in [Unicode Standard Annex #31](#).

NOTE 348 — The characters in `ID_Continue` are those in `ID_Start` together with those in the Unicode General Category classes “Mn”, “Mc”, “Nd”, and “Pc” together with those with the Unicode property `Other_ID_Continue` but none of which have the Unicode properties `Pattern_Syntax` or `Pattern_White_Space`. The characters in `XID_Continue` are derived from those in `ID_Continue` to ensure closure under NFKC normalization. Every character in `XID_Start` is also a character in `XID_Continue`.

- 3) The *representative form* *RF* of a <non-delimited identifier> *NDI* is the character string comprising the sequence of characters contained in *NDI*. *RF* shall not be the zero-length character string.

NOTE 349 — Provisions regarding the assumed Unicode normalization of character strings apply. See Subclause 4.16.3, “Character string types”.

- 4) The *representative form* *RF* of a <delimited identifier> *DI* is the character string specified by the <double quoted character sequence>, or the <accent quoted character sequence> contained in *DI*. *RF* shall not be the zero-length character string.

NOTE 350 — Provisions regarding the assumed Unicode normalization of character strings apply. See Subclause 4.16.3, “Character string types”.

- 5) For every <non-delimited identifier> or <delimited identifier> *NDIODI*:

- a) Let *RF* be the representative form of *NDIODI*.

- b) The maximum length in characters of *RF* shall be $2^{14} - 1 = 16383$.

NOTE 351 — This maximum length is modified by [Conformance Rule 1](#)).

- c) *RF* shall not contain characters of the Unicode General Category classes “Cc”, “Cf”, “Cn”, “Cs”, “Zl”, or “Zp”.

NOTE 352 — The Unicode General Category classes “Cc”, “Cf”, “Cn”, “Cs”, “Zl”, and “Zp” are assigned to Unicode characters that are, respectively, Control codes (other), Formal Control codes, Not Assigned codes, Surrogates, Line Separators, and Paragraph Separators.

- d) *RF* shall not contain characters of the Unicode General Category class “Zs” other than <space>.

NOTE 353 — Unicode General Category class “Zs” identifies characters that are Space Separators (whitespace).

- e) Whether characters of the Unicode General Category class “Co” are permitted to be contained in *RF* is implementation-defined (IA020).

NOTE 354 — The Unicode General Category class “Co” is assigned to private use characters.

- 6) <whitespace> is any consecutive sequence of Unicode characters with the property `White_Space`.

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

NOTE 355 — These are the characters the Unicode General Category classes “Zs”, “Zl” and “Zp” together with the characters: U+0009 (Horizontal Tabulation), U+000A (Line Feed), U+000B (Vertical Tabulation), U+000C (Form Feed), U+000D (Carriage Return), and U+0085 (Next Line).

- 7) <truncating whitespace> is an implementation-defined (IV023) subset of Unicode characters with the property White_Space; the subset shall always include at least <space>.

NOTE 356 — Since the Unicode definition of White_Space is subject to the addition of new characters, this definition prevents an existing conforming GQL-implementation from being made non-conforming by such a change. However, GQL-implementations are expected to align themselves with the most recent Unicode definition in a timely manner.

- 8) A <bidirectional control character> is any of the following characters of the Unicode General Category class “Cf”: LEFT-TO-RIGHT EMBEDDING (U+202A), RIGHT-TO-LEFT EMBEDDING (U+202B), POP DIRECTIONAL FORMATTING (U+202C), LEFT-TO-RIGHT OVERRIDE (U+202D), RIGHT-TO-LEFT OVERRIDE (U+202E), LEFT-TO-RIGHT ISOLATE (U+2066), RIGHT-TO-LEFT ISOLATE (U+2067), FIRST STRONG ISOLATE (U+2068), POP DIRECTIONAL ISOLATE (U+2069).

NOTE 357 — The Unicode General Category class “Cf” is assigned to Unicode characters that are Formal Control codes.

- 9) <simple comment character> is any character in the GQL source text character repertoire that is not a <newline>.
- 10) <bracketed comment contents> is any character string whose characters are in the GQL source text character repertoire and that does not contain <bracketed comment terminator>.
- 11) Whether a <simple comment character> or <bracketed comment contents> shall not be or contain, respectively, a <bidirectional control character> is implementation-defined (IA019).

NOTE 358 — This mitigates CVE-2021-42574.

- 12) <newline> is the implementation-defined (IA023) end-of-line indicator.

NOTE 359 — <newline> is typically represented by U+000A (“Line Feed”) and/or U+000D (“Carriage Return”); however, this representation is not required by this document.

- 13) A <token>, other than a <byte string literal> shall not contain a <separator>.
- 14) Any <token> may be followed by a <separator>. A <non-delimiter token> shall be followed by a <delimiter token> or a <separator>.
- NOTE 360 — If the Format does not allow a <non-delimiter token> to be followed by a <delimiter token>, then that <non-delimiter token> will be followed by a <separator>.

- 15) GQL source text containing one or more instances of <simple comment> is equivalent to the same GQL source text with each <simple comment> replaced with <newline>.

- 16) GQL source text containing one or more instances of <bracketed comment> is equivalent to the same GQL source text with each <bracketed comment> BC replaced with,

Case:

- a) If BC contains a <newline>, then <newline>.
- b) Otherwise, <space>.

- 17) For every <non-delimited identifier> NDI, there is exactly one corresponding *case-normal form* CNF. CNF is a character string derived from the representative form RF of NDI as follows.

Let N be the number of characters in RF. For i , $1 \text{ (one)} \leq i \leq N$, the i -th character M_i of RF is transliterated into the corresponding character or characters of CNF as follows.

Case:

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

- 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 .

NOTE 361 — Any lower-case letters for which there are no upper-case equivalents are left in their lower-case form.

NOTE 362 — The case-normal form of a <non-delimited identifier> is used in excluding <reserved word>s.

- 18) The case-normal form of a <non-delimited identifier> shall not be equal, according to the comparison rules in Subclause 19.3, "<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 a <character string literal>.

NOTE 363 — It is the intention that no keyword specified in this document or revisions thereto will end with an <underscore>.

- 19) Two <non-delimited identifier>s are equivalent if their representative forms compare equally according to the comparison rules in Subclause 19.3, "<comparison predicate>".
- 20) A <non-delimited identifier> and a <delimited identifier> are equivalent if their representative forms compare equally according to the comparison rules in Subclause 19.3, "<comparison predicate>".
- 21) Two <delimited identifier>s are equivalent if their representative forms compare equally according to the comparison rules in Subclause 19.3, "<comparison predicate>".
- 22) A *new system-generated regular identifier* is a new system-generated identifier that also conforms to the Format and Syntax Rules of <regular identifier>.
- 23) A *new system-generated identifier IDENT* that is possibly qualified by additional syntactic requirements is a system-generated identifier that is generated at a given point during the application *APP* of Rules for the GQL-program *PROGRAM* of the GQL-request such that:
 - a) *IDENT* conforms to the Format and Syntax Rules of <identifier>.
 - b) Within the context provided by *APP*, it holds that:
 - i) *IDENT* does not identify an existing named object or construct prior to being explicitly bound.
 - ii) *IDENT* adheres to all specified qualifying additional syntactic requirements
 - c) *IDENT* does not identify a session parameter in the current session context or a dynamic parameter in the current request context.
 - d) *IDENT* is not equivalent to an <identifier> or a <separated identifier> contained in *PROGRAM*.
 - e) *IDENT* is not equivalent to any other system-generated identifier obtained during the application of Rules for *PROGRAM*.
- 24) For every <delimited identifier> or <non-delimited identifier> *DIONDI*, there is exactly one *canonical name form* that is a character string derived from the representative form of *DIONDI* using an implementation-defined (IW023) mechanism such that the canonical name forms of equivalent identifiers are always identical.
- 25) For the purposes of identifying keywords, any <simple Latin lower-case letter> contained in a candidate <keyword> shall be effectively treated as the corresponding <simple Latin upper-case letter>.

ISO/IEC 39075:2024(en)
21.3 <token>, <separator>, and <identifier>

NOTE 364 — The equivalence of <simple Latin lower-case letter>s and <simple Latin upper-case letter> is specified in Subclause 21.4, “<GQL terminal character>”, Syntax Rule 3).

- 26) A *parameter reference* is either a <substituted parameter reference> or a <general parameter reference>.
- 27) The *parameter name* of parameter reference *PR* is the parameter name specified by the <parameter name> simply contained in *PR*.

General Rules

None.

Conformance Rules

- 1) Without Feature GB01, “Long identifiers”, in conforming GQL language, the maximum length in characters of the representative form of a <non-delimited identifier> or a <delimited identifier> shall be $2^7 - 1 = 127$.
- 2) Without Feature GB02, “Double minus sign comments”, conforming GQL language shall not contain a <simple comment introducer> that is a <double minus sign>.
- 3) Without Feature GB03, “Double solidus comments”, conforming GQL language shall not contain a <simple comment introducer> that is a <double solidus>.

21.4 <GQL terminal character>

Function

Define the terminal symbols of the GQL language.

Format

```
<GQL terminal character> ::=
  <GQL language character>
  | <other language character>

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

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

<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

<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

<hex digit> ::=
  <standard digit> | A | B | C | D | E | F | a | b | c | d | e | f

<digit> ::=
  <standard digit>
  | <other digit>

<standard digit> ::=
  <octal digit> | 8 | 9

<octal digit> ::=
  <binary digit> | 2 | 3 | 4 | 5 | 6 | 7

<binary digit> ::=
  0 | 1

<other digit> ::=
  !! See the Syntax Rules.

<GQL special character> ::=
  <space>
  | <ampersand>
  | <asterisk>
  | <colon>
  | <equals operator>
  | <comma>
  | <commercial at>
  | <dollar sign>
  | <double quote>
  | <exclamation mark>
  | <grave accent>
  | <right angle bracket>
```

ISO/IEC 39075:2024(en)
21.4 <GQL terminal character>

| <left brace>
| <left bracket>
| <left paren>
| <left angle bracket>
| <minus sign>
| <period>
| <plus sign>
| <question mark>
| <quote>
| <reverse solidus>
| <right brace>
| <right bracket>
| <right paren>
| <solidus>
| <underscore>
| <vertical bar>
| <percent>
| <tilde>

<space> ::=
!! U+0020

<ampersand> ::=
& !! U+0026

<asterisk> ::=
* !! U+002A

<colon> ::=
: !! U+003A

<comma> ::=
, !! U+002C

<commercial at> ::=
@ !! U+0040

<dollar sign> ::=
\$!! U+0024

<double quote> ::=
" !! U+0022

<equals operator> ::=
= !! U+003D

<exclamation mark> ::=
! !! U+0021

<right angle bracket> ::=
> !! U+003E

<grave accent> ::=
` !! U+0060

<left brace> ::=
{ !! U+007B

<left bracket> ::=
[!! U+005B

<left paren> ::=
(!! U+0028

<left angle bracket> ::=

ISO/IEC 39075:2024(en)
21.4 <GQL terminal character>

< !! U+003C

<minus sign> ::=
- !! U+002D

<percent> ::=
% !! U+0025

<period> ::=
. !! U+002E

<plus sign> ::=
+ !! U+002B

<question mark> ::=
? !! U+003F

<quote> ::=
' !! U+0027

<reverse solidus> ::=
\ !! U+005C

<right brace> ::=
} !! U+007D

<right bracket> ::=
] !! U+005D

<right paren> ::=
) !! U+0029

<solidus> ::=
/ !! U+002F

<tilde> ::=
~ !! U+007E

<underscore> ::=
_ !! U+005F

<vertical bar> ::=
| !! U+007C

<other language character> ::=
!! See the Syntax Rules.

Syntax Rules

- 1) <other language character> is any Unicode character not contained in <GQL language character>.
- 2) <other digit> is any Unicode character in the Unicode General Category class “Nd” not contained in <standard digit>.
- 3) 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 every *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>.

General Rules

None.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

22 Additional common rules

22.1 Annotation of a <GQL-program>

Function

Annotate a <GQL-program> with contextually provided metadata.

NOTE 365 — See Subclause 4.7.3, “Execution of GQL-requests”.

Subclause Signature

“Annotation of a <GQL-program>” [Syntax Rules] (
 Parameter: “GQL_PROGRAM”,
 Parameter: “INI_SCHEMA”,
 Parameter: “INI_GRAPH”,
 Parameter: “SES_PARAMS”,
 Parameter: “DYN_PARAMS”
)

GQL_PROGRAM — a <GQL-program>.

INI_SCHEMA — either “not set”, or an <absolute catalog schema reference>.

INI_GRAPH — either “not set” or a material graph reference value.

SES_PARAMS — a dictionary of general parameters.

DYN_PARAMS — a dictionary of general parameters.

Syntax Rules

- 1) Let *PROG* be the *GQL_PROGRAM*, let *IS* be the *INI_SCHEMA*, let *IG* be the *INI_GRAPH*, let *SESP* be the *SES_PARAMS*, and let *DYNP* be the *DYN_PARAMS* in an application of the Syntax Rules of this Subclause. .
- 2) *PROG* is annotated with an initial schema and an initial graph as follows:
 - a) The initial schema of *PROG* is *IS*.
 - b) The initial graph of *PROG* is *IG*.
- 3) For every <substituted parameter reference> *SPR* simply contained in *PROG*:
 - a) Let *PN* be the parameter name of *SPR*.
 - b) *PN* shall be the parameter name of a general parameter in *DYNP*.
 - c) Let *PV* be the parameter value of the general parameter in *DYNP* whose parameter name is *PN*.
 - d) *PV* shall be GQL source text.

ISO/IEC 39075:2024(en)
22.1 Annotation of a <GQL-program>

NOTE 366 — All comments are implicitly removed from GQL source text, as detailed by the Syntax Rules of Subclause 21.3, “<token>, <separator>, and <identifier>”. This enables later Syntax Rules to simply verify the Format of parameter substitutions without having to cater for comments.

e) *SPR* is annotated with a parameter name and a parameter substitution as follows:

- i) The parameter name of *SPR* is *PN*.
- ii) The parameter substitution of *SPR* is *PV*.

NOTE 367 — Parameter substitutions are substituted for certain BNF non-terminal instances and are required to conform to the Format of the corresponding BNF non-terminal symbols only. Consequently, GQL-implementations are required to ensure that parameter substitutions do not contain additional characters (such as unclosed <bracketed comment>s) to avoid potential security issues.

4) For every <session parameter specification> *SPS* simply contained in *PROG*:

- a) Let *GPR* be the <general parameter reference> simply contained in *SPS*.
- b) Let *PN* be the parameter name of *GPR*.
- c) *SPS* is annotated with a parameter name and a parameter value type as follows.

Case:

- i) If *PN* is the parameter name of a general parameter *SP* in *SESP*, then *SPS* is said to reference the defined session parameter *SP* and:
 - 1) The parameter name of *SPS* is *PN*.
 - 2) The parameter value type of *SPS* is the parameter value type of *SP*.
- ii) Otherwise, *SPS* does not reference a defined session parameter and:
 - 1) The parameter name of *SPS* is *PN*.
 - 2) The parameter value type of *SPS* is “not set”.

5) For every <dynamic parameter specification> *DPS* simply contained in *PROG*:

- a) Let *SSC* be the <session set command> that simply contains *DPS*.
- b) Let *GPR* be the <general parameter reference> simply contained in *DPS*.
- c) Let *PN* be the parameter name of *GPR*.
- d) *DPS* is annotated with a parameter name and a parameter value type as follows.

NOTE 368 — This is a recursive definition.

Case:

- i) If there are other <session set command>s that are simply contained in *PROG* that precede *SSC* in *PROG* and that simply contain a <session parameter specification> whose parameter name is *PN*, then:
 - 1) Let *DT* be the declared type of the <session parameter specification> simply contained in the rightmost such <session set command>.
 - 2) The parameter name of *DPS* is *PN*.
 - 3) The parameter value type of *DPS* is *DT*.
- ii) Otherwise:

ISO/IEC 39075:2024(en)
22.1 Annotation of a <GQL-program>

- 1) *PN* shall be the parameter name of a general parameter in *DYNP*.
- 2) Let *PVT* be the parameter value type of the general parameter in *DYNP* whose parameter name is *PN*.
- 3) If *PN* is the parameter name of a session parameter *SP* in *SESP*, then *PVT* shall be a subtype of the parameter value type of *SP*.
- 4) The parameter name of *DPS* is *PN*.
- 5) The parameter value type of *DPS* is *PVT*.

General Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

22.2 Machinery for graph pattern matching

Function

Define the infrastructure (alphabet, mappings and related definitions) used in graph pattern matching.

Subclause Signature

“Machinery for graph pattern matching” [General Rules] (
Parameter: “PROPERTY GRAPH”,
Parameter: “PATH PATTERN LIST”
) Returns: “MACHINERY”

PROPERTY GRAPH — a property graph.

PATH PATTERN LIST — a <path pattern list>.

MACHINERY — the machinery to be used for graph pattern matching.

Syntax Rules

None.

General Rules

- 1) Let *PG* be the *PROPERTY GRAPH* and let *PPL* be the *PATH PATTERN LIST* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *MACHINERY*.
- 2) Let *SVV* be the set of names of node variables declared in *PPL* at the same depth of graph pattern matching, and let *SEV* be the set of names of edge variables declared in *PPL* at the same depth of graph pattern matching.
- 3) For each subpath variable *SPV* declared in *PPL* at the same depth of graph pattern matching, let *SPVBEGIN* and *SPVEND* be two distinct <identifier>s that are distinct from every <identifier> in *SVV* $\cup$ *SEV* and from every <identifier> created by this rule. *SPVBEGIN* is the *begin subpath symbol* and *SPVEND* is the *end subpath symbol* associated with *SPV*. Let *SPS* be the set of every subpath symbol.
- 4) Let *[]* and *-* be mutually distinct <identifier>s that are distinct from every <identifier> in *SVV* $\cup$ *SEV* $\cup$ *SPS*. These are, respectively, the *anonymous node symbol* and the *anonymous edge symbol*. Let *SAS* be the set of anonymous symbols.
- 5) Let *NPP* be the number of <parenthesized path pattern expression>s contained in *PPL* at the same depth of graph pattern matching. Let *PPPE*₁, ..., *PPPE*_{*NPP*} be an enumeration of the <parenthesized path pattern expression>s contained in *PPL* at the same depth of graph pattern matching. For every *i*, 1 (one) $\leq i \leq NPP$, *i* is the *bracket index* of *PPPE*_{*i*}.
- 6) Let *[*₁*]*, ..., *[*_{*NPP*}*]*, *]*₁, ..., *]*_{*NPP*} be 2 * *NPP* <identifier>s that are mutually distinct, and distinct from every member of *SVV* $\cup$ *SEV* $\cup$ *SPS* $\cup$ *SAS* and from every graph element of *PG*. These are called *bracket symbols*. For every bracket symbols *[*_{*j*}*]* or *]*_{*j*}, *j* is the *bracket index* of the bracket symbol. There are two bracket symbols for each bracket index *j* between 1 (one) and *NPP*, corresponding to the <parenthesized path pattern expression>s *PPPE*_{*j*}. Let *SBS* be the set of bracket symbols.

ISO/IEC 39075:2024(en)
22.2 Machinery for graph pattern matching

- 7) Let GX be the set whose members are the graph elements of PG , the subpath symbols, and the bracket symbols.
- 8) Let ABC be $SPS \cup SBS \cup SVV \cup SEV \cup SAS$. ABC is the *alphabet*. The members of ABC are *symbols*.
- 9) A *word* is a string of elements of ABC .
- 10) An *elementary binding* is a pair (LET, GE) where LET is a member of ABC and GE is a member of GX , such that:

Case:

- a) If LET is a bracket symbol, then $LET = GE$. In this case, the elementary binding is a *bracket symbol binding*. If LET is a start bracket symbol, then the elementary binding is a *start bracket symbol binding*; otherwise, it is an *end bracket symbol binding*. The bracket index of LET is the bracket index of the bracket symbol binding.
 - b) If LET is a subpath symbol, then $LET = GE$. In this case, the elementary binding is a *subpath symbol binding*.
 - c) If LET is the name of a node variable or the anonymous node symbol, then GE is a node. In this case the elementary binding is a *node symbol binding*.
 - d) If LET is the name of an edge variable or the anonymous edge symbol, then GE is an edge. In this case, the elementary binding is an *edge symbol binding*.
- 11) If $EB = (LET, GE)$ is an elementary binding, then EB is an *elementary binding of LET* , and EB binds LET to GE .

NOTE 369 — An elementary binding is a mapping of a symbol (an <identifier>) to a member of GX ; it is not the binding of a graph pattern variable. In particular, if LET is the name of an element variable EV , there can be more than one elementary binding of LET in a multi-path binding. In a consistent path binding (defined subsequently), two elementary bindings of LET necessarily bind to the same graph element in contexts in which LET is exposed as unconditional singleton; otherwise, elementary bindings of LET are independent of one another, and can bind to more than one graph element. Similarly, references to EV are context-dependent, and can resolve to a list that is a proper subset of all the graph elements bound to LET by elementary bindings. Resolution of <element variable reference>s is performed by the General Rules of Subclause 22.6, “Application of bindings to evaluate an expression”.

- 12) When an elementary binding (LET, GE) binds LET to a graph element GE during the evaluation of an <element pattern where clause>, a new field whose name is LET and whose value is a graph element reference value to GE is temporarily added to the current working record until the evaluation has finished.
- 13) A *path binding* is a sequence of zero or more elementary bindings, $B = (LET_1, GE_1), \dots (LET_N, GE_N)$. Given a path binding B :

NOTE 370 — Unlike the definition of path, the definition of path binding allows a sequence of zero elementary bindings. For example, a quantifier can iterate 0 (zero) times, resulting in an empty path binding. The result of a <path pattern>, on the other hand, is unable to be empty because of a Syntax Rule that enforces a minimum node count of 1 (one).

- a) The *word* of B is the sequence $LET_1, \dots, LET_N$.
- b) The *annotated path* of B is the sequence $GE_1, \dots, GE_N$.

NOTE 371 — If the path binding is consistent (defined subsequently), then the annotated path of B contains within it a path that matches the word of B , plus mark-up with bracket symbols and subpath symbols indicating how to interpret the path as a match to the word.

- c) The *compressed path binding CPB* of B is obtained from B as follows:
 - i) Let CPB be a copy of B .

ISO/IEC 39075:2024(en)
22.2 Machinery for graph pattern matching

- ii) All subpath symbol bindings and all bracket symbol bindings are deleted from *CPB*.
- iii) Each maximal subsequence *MS* of *CPB* comprising one or more consecutive node bindings is replaced by a single compressed binding, whose components are the following:
 - 1) The first component is a list of the names of node variables in the first component of the node bindings of *MS*.
 NOTE 372 — This list is empty if only the anonymous node symbol is bound in *MS*.
 - 2) The second component is the node that is bound by the first node binding in *MS*.
 NOTE 373 — For consistent path bindings, consecutive node bindings will bind the same node.
- iv) In each edge binding *EB* of *CPB*, the first component is replaced by a list of zero or one name of an edge variable, retaining the name of the edge variable in the first component of *EB*, if any.

- d) The *extracted path* *XP* of *B* is obtained from the compressed path binding of *B* as the sequence of the second components of the compressed bindings of *CPB*.

NOTE 374 — The extracted path is a path if the path binding is consistent and non-empty; otherwise, the extracted path is not necessarily a path.

- 14) Let *REDUCE* be a function that maps path bindings to path bindings, determined as follows:

- a) Let *PBIN* be a path binding.
- b) Let *PBOUT* be a copy of *PBIN*.
- c) All bracket bindings are removed from *PBOUT*.
- d) Every element binding to a temporary node variable is replaced by an anonymous node binding to the same node in *PBOUT*.
- e) Every element binding to a temporary edge variable is replaced by an anonymous edge binding to the same edge in *PBOUT*.
- f) The following steps are performed on *PBOUT* repeatedly until no more anonymous node bindings can be removed:
 - i) If there are two adjacent anonymous node bindings, then the second is removed.
 - ii) If there is a binding of a node variable adjacent to an anonymous node binding, then the anonymous node binding is removed.
- g) *REDUCE(PBIN)* is *PBOUT*.

- 15) In a path binding, two node bindings are *separable* if there is an edge binding between them.

- 16) A path binding *B* is *consistent* if all of the following conditions are true:

- a) The extracted path *XP* of *B* is either the empty sequence or a path of *PG*.

NOTE 375 — That is, either *XP* is empty or *XP* begins with a node, alternates between nodes and edges, each edge in *XP* connects the node before and after it, and *XP* ends with a node.

- b) For every two node bindings (*LET*₁, *GE*₁) and (*LET*₂, *GE*₂) that are not separable, *GE*₁ = *GE*₂.

NOTE 376 — If there are only bracket symbol or subpath symbol bindings between two node bindings, then the node bindings must bind to the same node.

ISO/IEC 39075:2024(en)
22.2 Machinery for graph pattern matching

- c) For every edge binding $EB = (LET, GE)$, let (LET_LEFT, GE_LEFT) be the last node binding to the left of EB and let (LET_RIGHT, GE_RIGHT) be the first node binding to the right of EB .

Case:

- i) If GE is an undirected edge, then GE is an edge connecting GE_LEFT and GE_RIGHT .
- ii) If GE is a directed edge, then either GE_LEFT is the source node and GE_RIGHT is the destination node of GE , or GE_RIGHT is the source node and GE_LEFT is the destination node of GE .

NOTE 377 — The directionality constraint of the edge binding is fully checked during the generation of the regular language for <path concatenation>.

- d) For every start bracket symbol binding SBS contained in B , let j be the bracket index of SBS . All of the following are true:

- i) There is an end bracket symbol binding contained in B and following SBS whose bracket index is j . Let EBS be the first such end bracket symbol binding. Let PPS be the explicit or implicit <path mode prefix> simply contained in the <parenthesized path pattern expression> whose bracket index is j . Let PM be the <path mode> contained in PPS .

ii) Case:

- 1) If PM is TRAIL, then there is no pair of edge bindings at two different positions between SBS and EBS that bind the same edge.

NOTE 378 — This definition does not take note of the symbols that are bound, only the edges. It is a violation of the TRAIL <path mode> if the symbols in the pair of distinct edge bindings are both anonymous edge symbols, are both edge variables (whether the same or different), or one is the anonymous edge symbol and the other is an edge variable.

- 2) If PM is SIMPLE, then no two separable node bindings between SBS and EBS bind the same node, except that the first and last node binding between SBS and EBS may bind the same node.

NOTE 379 — This definition does not take note of the symbols that are bound, only the nodes. It is a violation of the SIMPLE <path mode> if the symbols in the pair of separable node bindings are both anonymous node symbols, are both node variables (whether the same or different), or one is the anonymous node symbol and the other is a node variable. However, the first and last node binding between SBS and EBS can bind the same node without violating the SIMPLE <path mode>.

- 3) If PM is ACYCLIC, then no two separable node bindings between SBS and EBS bind the same node.

NOTE 380 — This definition does not take note of the symbols that are bound, only the nodes. It is a violation of the ACYCLIC <path mode> if the symbols in the pair of separable node bindings are both anonymous node symbols, both are node variables (whether the same or different), or one is the anonymous node symbol and the other is a node variable.

NOTE 381 — The <path mode> WALK imposes no constraints on the extracted path.

- 17) A multi-path binding is an n -tuple $(PB_1, \dots, PB_N)$ for some positive integer N such that each PB_i , $1 \leq i \leq N$, is a path binding.

- 18) If S and T are sets of strings, then let $S \cdot T$ be the set of strings formed by concatenating an element of S followed by an element of T ; that is, $S \cdot T = \{ s t \mid s \text{ is an element of } S, t \text{ is an element of } T \}$.

NOTE 382 — The $\cdot$ operator will be used to concatenate words (strings of symbols) and path bindings (strings of elementary bindings).

ISO/IEC 39075:2024(en)
22.2 Machinery for graph pattern matching

- 19) If S is a set of strings, then let S^0 be the set whose only element is the string of length 0 (zero), and for each non-negative integer n , let S^{n+1} be $S^n \cdot S$. Let S^* be the union of S^n for every non-negative integer n .

NOTE 383 — If S is non-empty, then S^* is an infinite set. However, a finite result for every <graph pattern> is assured by the syntactic requirement that every <quantified path primary> is bounded, contained in a restrictive <parenthesized path pattern expression>, contained in a selective <path pattern>, or in the scope of the <different edges match mode>.

- 20) *REDUCE* is extended to multi-path bindings as follows. If $MPB = (PB_1, \dots, PB_n)$ is a multi-path binding, then $REDUCE(MPB) = (REDUCE(PB_1), \dots, REDUCE(PB_n))$.
- 21) Let *MACH* be a data structure comprising the following:
- a) *ABC*, the alphabet, formed as the disjoint union of the following:
 - i) *SVV*, the set of names of node variables.
 - ii) *SEV*, the set of names of edge variables.
 - iii) *SPS*, the set of subpath symbols.
 - iv) *SAS*, the set of anonymous symbols.
 - v) *SBS*, the set of bracket symbols.
 - b) *REDUCE*, the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.
- 22) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *MACH* as *MACHINERY*.

Conformance Rules

None.

22.3 Evaluation of a <path pattern expression>

Function

Evaluate a <path pattern expression>.

Subclause Signature

"Evaluation of a <path pattern expression>" [General Rules] (
Parameter: "PROPERTY GRAPH",
Parameter: "PATH PATTERN LIST",
Parameter: "MACHINERY",
Parameter: "SPECIFIC BNF INSTANCE"
) Returns: "SET OF MATCHES"

PROPERTY GRAPH — a property graph.

PATH PATTERN LIST — a <path pattern list>.

MACHINERY — the machinery for graph pattern matching.

SPECIFIC BNF INSTANCE — the specific BNF non-terminal instance to be evaluated.

SET OF MATCHES — the set of local matches to the SPECIFIC BNF INSTANCE.

Syntax Rules

None.

General Rules

- 1) Let *PG* be the *PROPERTY GRAPH*, let *PPL* be the *PATH PATTERN LIST*, let *MACH* be the *MACHINERY*, and let *SBI* be the *SPECIFIC BNF INSTANCE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *SET OF MATCHES*.
- 2) The following components of *MACH* are identified:
 - a) *ABC*, the alphabet, formed as the disjoint union of the following:
 - i) *SVV*, the set of names of node variables.
 - ii) *SEV*, the set of names of edge variables.
 - iii) *SPS*, the set of subpath symbols.
 - iv) *SAS*, the set of anonymous symbols.
 - v) *SBS*, the set of bracket symbols.
 - b) *REDUCE*, the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.
- 3) For every BNF non-terminal instance *BNT* that is a <path pattern expression>, <path term>, <path pattern union>, <path factor>, <path concatenation>, <path primary>, <quantified path primary>, <questioned path primary>, <element pattern>, or <parenthesized path pattern expression> equal to or contained in *SBI* at the same depth of graph pattern matching, the following are defined by

ISO/IEC 39075:2024(en)
22.3 Evaluation of a <path pattern expression>

simultaneous recursion: the *regular language of BNT*, denoted $RL(BNT)$, defined as a set of words over ABC ; and the *set of local matches to BNT*, denoted $SLM(BNT)$, defined as a set of path bindings.

NOTE 384 — $SLM(BNT)$ is consistent except when concatenating an edge pattern prior to concatenating the following node pattern. Restrictive path modes are enforced when generating $SLM(BNT)$ for the <parenthesized path pattern expression> that declares the path mode.

NOTE 385 — $RL(BNT)$ and $SLM(BNT)$ can be infinite sets if BNT contains an effectively unbounded quantifier. Every effectively unbounded quantifier is required to be either contained in a selective <path pattern> or in the scope of the <different edges match mode>. For a selective <path pattern> the potentially infinite set of local matches is subsequently reduced to a finite set by the General Rules of [Subclause 22.4, "Evaluation of a selective <path pattern>"](#). For <different edges match mode>, the matches are reduced to a finite set by the General Rules of [Subclause 16.4, "<graph pattern>"](#).

NOTE 386 — The BNF non-terminal symbols are listed above in the "top down" order of appearance in the Format of [Subclause 16.7, "<path pattern expression>"](#); the definitions in the following subrules treat instances of the same BNF non-terminal symbols in "bottom up" order.

Case:

- a) If BNT is a <parenthesized path pattern expression>, then let PPE be the <path pattern expression> immediately contained in BNT and let j be the bracket index of BNT .

NOTE 387 — "Bracket index" is defined in [Subclause 22.2, "Machinery for graph pattern matching"](#).

i) Case:

- 1) If BNT simply contains a <subpath variable declaration> SVD , then let SV be the subpath variable declared by SVD . Let BSV be the begin subpath symbol associated with SV and let ESV be the end subpath symbol associated with SV . Let $BSVBINDING$ be (BSV, BSV) and let $ESVBINDING$ be (ESV, ESV) .
- 2) Otherwise, let BSV , ESV , $BSVBINDING$ and $ESVBINDING$ be the empty string.

ii) $RL(BNT)$ is $\{ \text{'[j' } \} \cdot \{ BSV \} \cdot RL(PPE) \cdot \{ ESV \} \cdot \{ \text{']j' } \}$

iii) Let $STPB$ be $SLM(PPE)$.
Let $SLMMAYBE$ be $\{ \{ \text{'[j' , ']j' } \} \cdot \{ BSVBINDING \} \cdot STPB \cdot \{ ESVBINDING \} \cdot \{ \text{'[j' , ']j' } \} \}$

$SLM(BNT)$ is the set of every path binding in $SLMMAYBE$ that is consistent.

NOTE 388 — That is, the words in $RL(BNT)$ are formed by surrounding the words of $RL(PPE)$ by the bracket symbols '[j' and ']j' . If BNT contains a subpath variable declaration, then the words of $RL(BNT)$ are also surrounded by the begin and end subpath symbol associated with that subpath variable. Similarly, the path bindings in $SLMMAYBE$ are formed by surrounding the path bindings with bracket bindings and, if there is a subpath declaration, with the corresponding begin and end subpath bindings. Eliminating inconsistent bindings from $SLMMAYBE$ to get $SLM(BNT)$ has the effect of enforcing restrictive path modes.

- b) If BNT is an <element pattern> EP , then:

i) Case:

- 1) If EP declares an element variable EV , then let EPI be the name of EV .
- 2) If EP is a <node pattern>, then let EPI be '()' , the anonymous node symbol.
- 3) If EP is an <edge pattern>, then let EPI be '-' , the anonymous edge symbol.

ii) $RL(BNT)$ is $\{ EPI \}$, the set whose sole member is EPI .

iii) $SLM(BNT)$ is the set of every elementary binding (EPI, GE) such that:

- 1) If EP is a <node pattern>, then GE is a node.

ISO/IEC 39075:2024(en)
22.3 Evaluation of a <path pattern expression>

- 2) If EP is an <edge pattern>, then GE is an edge.
 - 3) If EP simply contains a <label expression> LE , then $True$ is the *TRUTH VALUE* returned as TV when the Syntax Rules of Subclause 22.5, “Satisfaction of a <label expression> by a label set”, with LE as *LABEL EXPRESSION* and the label set of GE as *LABEL SET*.
- c) If BNT is a <quantified path primary>, then:
- i) Let PP be the <path primary> immediately contained in BNT . As a result of the transformations in the Syntax Rules, PP is a <parenthesized path pattern expression>. Let R be $RL(PP)$ and let S be $SLM(PP)$.
 - ii) Let GQ be the <general quantifier> immediately contained in BNT .
 - iii) Let LB be the value of the <lower bound> contained in GQ .
 - iv) Case:
 - 1) If GQ contains an <upper bound>, then let UB be the value of the <upper bound>.
 - A) $RL(BNT)$ is $R^{LB} \cup R^{LB+1} \cup \dots \cup R^{UB-1} \cup R^{UB}$.
 - B) Let $TOOMUCH$ be $S^{LB} \cup S^{LB+1} \cup \dots \cup S^{UB-1} \cup S^{UB}$.
 - C) $SLM(BNT)$ is the set of those path bindings in $TOOMUCH$ that are consistent.
 - 2) Otherwise,
 - A) $RL(BNT)$ is $R^{LB} \cdot R^*$.
 - B) Let $WAYTOOMUCH$ be $S^{LB} \cdot S^*$.
 - C) $SLM(BNT)$ is the set of those path bindings in $WAYTOOMUCH$ that are consistent.
- d) If BNT is a <questioned path primary>, then:
- i) Let PP be the <path primary> immediately contained in BNT . As a result of the transformations in the Syntax Rules, PP is a <parenthesized path pattern expression>. Let R be $RL(PP)$ and let S be $SLM(PP)$.
 - ii) $RL(BNT)$ is $R^0 \cup R$.
 - iii) $SLM(BNT)$ is $S^0 \cup S$.
- e) If BNT is a <path primary>, then let $BNT2$ be the <element pattern> or <parenthesized path pattern expression> that is immediately contained in BNT .
- i) $RL(BNT)$ is $RL(BNT2)$.
 - ii) $SLM(BNT)$ is $SLM(BNT2)$.
- f) If BNT is a <path concatenation>, then:
- i) Let PST be the <path term> and let PC be the <path factor> that are immediately contained in BNT .
 - ii) $RL(BNT)$ is $RL(PST) \cdot RL(PC)$.
 - iii) Case:

ISO/IEC 39075:2024(en)
22.3 Evaluation of a <path pattern expression>

- 1) If PC is an <edge pattern>, then $SLM(BNT)$ is $SLM(PST) \cdot SLM(PC)$

NOTE 389 — If PC is an <edge pattern>, then there is a <node pattern> to its right, which will be concatenated in a subsequent iteration of this recursion; consistency, including the directionality constraint implied by the <edge pattern>, will be checked at that point.

- 2) If PC is a <node pattern>, and the last <element pattern> EP of PST is an <edge pattern>, then:

NOTE 390 — By transformations in the Syntax Rules of [Subclause 16.7, “<path pattern expression>”](#), an edge binding is always immediately preceded and followed by a node binding. The current rule handles the situation in which the edge has been bound as the last elementary binding of PST , and therefore PC is the node binding that immediately follows the edge binding. Also note that the Syntax Rules have transformed every <abbreviated edge pattern> to a <full edge pattern>.

- A) Let $SLMCONCAT$ be $SLM(PST) \cdot SLM(PC)$.
- B) For each path binding PB contained in $SLMCONCAT$,
- I) Let GE_RIGHT be the node that is bound in the last elementary binding of PB .
 - II) Let GE be the edge that is bound in the penultimate elementary binding of PB .
 - III) Let GE_LEFT be the node that is bound in the antepenultimate elementary binding of PB .
- C) Let the propositions L , U , and R be defined as follows:
- I) Proposition L is true if GE is a directed edge, GE_LEFT is the destination node of GE and GE_RIGHT is the source node of GE .
 - II) Proposition U is true if GE is an undirected edge, and GE_LEFT and GE_RIGHT are the nodes connected by GE .
 - III) Proposition R is true if GE is a directed edge, GE_LEFT is the source node of GE and GE_RIGHT is the destination node of GE .
- D) The *directionality constraint* of EP is
- Case:
- I) If EP is a <full edge pointing left>, then proposition L is true.
 - II) If EP is a <full edge undirected>, then proposition U is true.
 - III) If EP is a <full edge pointing right>, then proposition R is true.
 - IV) If EP is a <full edge left or undirected>, then proposition L , or proposition U , is true.
 - V) If EP is a <full edge undirected or right>, then proposition U , or proposition R , is true.
 - VI) If EP is a <full edge left or right>, then proposition L , or proposition R , is true.
 - VII) If EP is a <full edge any direction>, then at least one of proposition L , proposition U , or proposition R , is true.

ISO/IEC 39075:2024(en)
22.3 Evaluation of a <path pattern expression>

- E) $SLM(BNT)$ is the set of those path bindings of $SLMCONCAT$ that are consistent and satisfy the directionality constraint of EP .
- 3) Otherwise, $SLM(BNT)$ is the set of those path bindings in $SLM(PST) \cdot SLM(PC)$ that are consistent.
- g) If BNT is a <path factor>, then let $BNT2$ be the <path primary>, <quantified path primary> or <questioned path primary> immediately contained in BNT .
- i) $RL(BNT)$ is $RL(BNT2)$.
- ii) $SLM(BNT)$ is $SLM(BNT2)$.
- h) If BNT is a <path pattern union>, then let NMA be the number of <path term>s immediately contained in BNT . Let $PMO_1, \dots, PMO_{NMA}$ be these <path term>s.
- i) $RL(BNT)$ is $RL(PMO_1) \cup RL(PMO_2) \cup \dots \cup RL(PMO_{NMA})$.
- ii) $SLM(BNT)$ is $SLM(PMO_1) \cup SLM(PMO_2) \cup \dots \cup SLM(PMO_{NMA})$.
- i) If BNT is <path term>, then let $BNT2$ be the <path factor> or <path concatenation> immediately contained in BNT .
- i) $RL(BNT)$ is $RL(BNT2)$.
- ii) $SLM(BNT)$ is $SLM(BNT2)$.
- j) If BNT is a <path pattern expression>, then let $BNT2$ be the <path term> or <path pattern union> immediately contained in BNT .
- NOTE 391 — <path multiset alternation> is transformed into <path pattern union> in the Syntax Rules and therefore it is not considered separately here.
- i) $RL(BNT)$ is $RL(BNT2)$.
- ii) $SLM(BNT)$ is $SLM(BNT2)$.
- 4) Let SM be $SLM(SBI)$.
- 5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives SM as *SET OF MATCHES*.

Conformance Rules

None.

22.4 Evaluation of a selective <path pattern>

Function

Evaluate a <path pattern> with a selective <path search prefix>.

Subclause Signature

"Evaluation of a selective <path pattern>" [General Rules] (

Parameter: "PROPERTY GRAPH",
Parameter: "PATH PATTERN LIST",
Parameter: "MACHINERY",
Parameter: "SELECTIVE PATH PATTERN",
Parameter: "INPUT SET OF LOCAL MATCHES"
) Returns: "OUTPUT SET OF LOCAL MATCHES"

PROPERTY GRAPH — a property graph.

PATH PATTERN LIST — a <path pattern list>.

MACHINERY — the machinery for graph pattern matching.

SELECTIVE PATH PATTERN — a selective <path pattern>.

INPUT SET OF LOCAL MATCHES — a set, possibly infinite, of matches to SELECTIVE PATH PATTERN.

OUTPUT SET OF LOCAL MATCHES — finite subset of the INPUT SET OF LOCAL MATCHES, selected according to the criterion indicated by the selective <path search prefix> of SELECTIVE PATH PATTERN.

Syntax Rules

None.

General Rules

1) Let *PG* be the *PROPERTY GRAPH*, let *PPL* be the *PATH PATTERN LIST*, let *MACH* be the *MACHINERY*, let *SEL* be the *SELECTIVE PATH PATTERN*, and let *INSLM* be the *INPUT SET OF LOCAL MATCHES* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *OUTPUT SET OF LOCAL MATCHES*.

2) It is implementation-defined (IA013) whether the General Rules of this Subclause are terminated if an exception condition is raised. If a GQL-implementation defines that it terminates execution because of an exception condition, it is implementation-dependent (UA006) which of the members of *CANDIDATES* (defined subsequently) are actually probed to establish whether they might raise an exception.

NOTE 392 — *CANDIDATES* is potentially an infinite set, but there are algorithms to enumerate this set so as to satisfy the selection criterion of the selective <path pattern> without testing all candidate solutions. Even if the GQL-implementation defines that it terminates when an exception condition is encountered on a particular candidate solution, the order of enumerating the candidates is implementation-dependent, and it is possible that a candidate solution that would raise an exception is never tested.

3) The following components of *MACH* are identified:

a) *ABC*, the alphabet, formed as the disjoint union of the following:

ISO/IEC 39075:2024(en)
22.4 Evaluation of a selective <path pattern>

- i) *SVV*, the set of names of node variables.
- ii) *SEV*, the set of names of edge variables.
- iii) *SPS*, the set of subpath symbols.
- iv) *SAS*, the set of anonymous symbols.
- v) *SBS*, the set of bracket symbols.
- b) *REDUCE*, the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.

4) Let *NP* be the number of <path pattern>s in *PPL*.

5) *SEL* is a selective <path pattern>. Let *j* be the bracket index of the <parenthesized path pattern expression> simply contained in *SEL*.

NOTE 393 — “Bracket index” is defined in Subclause 22.2, “Machinery for graph pattern matching”.

NOTE 394 — By a syntactic transformation in Subclause 16.4, “<graph pattern>”, this <parenthesized path pattern expression> is the entire content of *SEL* except possibly the declaration of a path variable.

6) Let *PSP* be the <path search prefix> simply contained in *SEL*.

7) Let *N* be the value of the <number of paths> or the <number of groups> specified in *PSP*. If *N* is not a positive integer, then an exception condition is raised: *data exception — invalid number of paths or groups (22G0F)* and no further General Rules of this Subclause are applied.

8) Let *p* be such that *SEL* is the *p*-th <path pattern> of *PPL*.

9) Let *CANDIDATES* be the set of every path binding *PBX* in *INSLM* such that all of the following conditions are true:

- a) For every unconditional singleton <element variable> *EV* exposed by *SEL*, *EV* is bound to a unique graph element by the elementary bindings of *EV* contained in *PBX*.

NOTE 395 — Anonymous symbols are not <element variable>s; there is no requirement that two anonymous symbols bind to the same graph element.

- b) For every <parenthesized path pattern expression> *PPPE* equal to or contained in *SEL*, let *i* be the bracket index of *PPPE*, and let ‘*i*’ and ‘*i*’ be the bracket symbols associated with *PPPE*. A *binding* of *PPPE* is a substring of *PBX* that begins with the bracket binding (‘*i*’, ‘*i*’) and ends with the next bracket binding (‘*i*’, ‘*i*’).

NOTE 396 — “Bracket index” is defined in Subclause 22.2, “Machinery for graph pattern matching”.

For every binding *BPPPE* of *PPPE* contained in *PBX*, all of the following are true:

- i) For every <element variable> *EV* that is exposed as an unconditional singleton by *PPPE*, *EV* is bound to a unique graph element by the elementary bindings of *EV* contained in *BPPPE*.

NOTE 397 — Anonymous symbols are not <element variable>s; there is no requirement that two anonymous symbols bind to the same graph element.

- ii) If *PPPE* contains a <parenthesized path pattern where clause> *PPPWC*, then *True* is the *VALUE* returned as *V* when the General Rules of Subclause 22.6, “Application of bindings to evaluate an expression”, with *PPL* as *GRAPH PATTERN*, the <search condition> simply contained in *PPPWC* as *EXPRESSION*, *MACH* as *MACHINERY*, *PBX* as *MULTI-PATH BINDING*, and a reference to *BPPPE* as a subset of *PBX* as *REFERENCE TO LOCAL CONTEXT*.

ISO/IEC 39075:2024(en)
22.4 Evaluation of a selective <path pattern>

NOTE 398 — This is the juncture at which an exception condition might be raised. It is implementation-defined whether to terminate if an exception condition is raised. The order of enumerating the members of *CANDIDATES* is implementation-dependent, and there is no requirement that a GQL-implementation test all candidate solutions, which can be an infinite set in any case.

- 10) Each path binding *PBX* of *CANDIDATES* is replaced by *REDUCE(PBX)*.
- 11) Redundant duplicate path bindings are removed from *CANDIDATES*.
- 12) *CANDIDATES* is partitioned as follows. For every path binding *PBX* in *CANDIDATES*, the partition of *PBX* is the set of every path binding *PBY* in *CANDIDATES* such that the first and last node bindings of *PBX* bind the same nodes as the first and last node bindings, respectively, of *PBY*.
- 13) Each partition *PART* of *CANDIDATES* is modified as follows.
Case:
 - a) If *PSP* is an <any path search>, then
Case:
 - i) If the number of path bindings in *PART* is *N* or less, then the entire partition *PART* is retained.
 - ii) Otherwise, it is implementation-dependent (UA005) which *N* path bindings of *PART* are retained.
 - b) If *PSP* is a <shortest path search>, then
Case:
 - i) If *PSP* is a <counted shortest path search>, then
Case:
 - 1) If the number of path bindings in *PART* is *N* or less, then the entire partition *PART* is retained.
 - 2) Otherwise, the path bindings of *PART* are sorted in increasing order of number of edges; the order of path bindings that have the same number of edges is implementation-dependent (US005). The first *N* path bindings in *PART* are retained.
 - ii) If *PSP* is <counted shortest group search>, then the path bindings in *PART* are grouped, with each group comprising those path bindings having the same number of edges. The groups are ordered in increasing order by the number of edges.
Case:
 - 1) If the number of groups in *PART* is *N* or less, then the entire partition *PART* is retained.
 - 2) Otherwise, the path bindings comprising the first *N* groups of *PART* are retained.
- 14) Let *OUTSLM* be the set of path bindings retained in *CANDIDATES* after the preceding modifications to its partitions.
- 15) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *OUTSLM* as *OUTPUT SET OF LOCAL MATCHES*.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

22.5 Satisfaction of a <label expression> by a label set

Function

Determine if a label set satisfies a <label expression>.

Subclause Signature

"Satisfaction of a <label expression> by a label set" [Syntax Rules] (
Parameter: "LABEL EXPRESSION",
Parameter: "LABEL SET"
) Returns: "TRUTH VALUE"

LABEL EXPRESSION — a <label expression>.

LABEL SET — the label set of a graph element.

TRUTH VALUE — True if the DEFINED LABEL SET satisfies the LABEL EXPRESSION; otherwise, False.

Syntax Rules

- 1) Let *LEXP* be the *LABEL EXPRESSION* and let *LS* be the *LABEL SET* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *TRUTH VALUE*.
- 2) A label set *LS* satisfies a <label expression> *LE* according to the following recursive definition:
 - a) If *LE* is a <label name> *L2*, then *L2* is a member of *LS*.
 - b) If *LE* is a <wildcard label>, then *LS* is non-empty.
NOTE 399 — This condition is always true; every label set is non-empty. The rule is written this way in case empty label sets are permitted in the future, or for guidance to a GQL-implementation that supports graph elements with no labels.
 - c) If *LE* is a <parenthesized label expression> *PLE*, then let *LE2* be the <label expression> simply contained in *PLE*; *LS* satisfies *LE2*.
 - d) If *LE* is a <label negation>, then let *LP* be the <label primary> simply contained in *LE*; *LS* does not satisfy *LP*.
 - e) If *LE* is a <label conjunction>, then let *L1* be the <label term> and let *L2* be the <label factor> simply contained in *LE*; *LS* satisfies *L1* and *LS* satisfies *L2*.
 - f) If *LE* is a <label disjunction>, then let *L1* be the <label expression> and let *L2* be the <label term> simply contained in *LE*; *LS* satisfies *L1* or *LS* satisfies *L2*.
- 3) Let *TV* be:
Case:
 - a) If *LS* satisfies *LEXP*, then True.
 - b) Otherwise, False.
- 4) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *TV* as *TRUTH VALUE*.

General Rules

None.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

22.6 Application of bindings to evaluate an expression

Function

Evaluate a <value expression> or <search condition> using the bindings of graph pattern variables determined by a local context within a multi-path binding.

Subclause Signature

"Application of bindings to evaluate an expression" [General Rules] (

Parameter: "GRAPH PATTERN",

Parameter: "EXPRESSION",

Parameter: "MACHINERY",

Parameter: "MULTI-PATH BINDING",

Parameter: "REFERENCE TO LOCAL CONTEXT"

) Returns: "VALUE"

GRAPH PATTERN — a <graph pattern>.

EXPRESSION — a <value expression> or <search condition>.

MACHINERY — the machinery for graph pattern matching.

MULTI-PATH BINDING — multi-path binding to the <graph pattern> in which to evaluate the expression.

REFERENCE TO LOCAL CONTEXT — an indication of a subset of the multi-path binding, the local context. Group bindings are confined to the local context; singleton bindings may look outside the local context.

VALUE — the evaluated value of EXPRESSION.

Syntax Rules

None.

General Rules

- 1) Let *GP* be the *GRAPH PATTERN*, let *EXP* be the *EXPRESSION*, let *MACH* be the *MACHINERY*, let *MPB* be the *MULTI-PATH BINDING*, and let *RTLCL* be the *REFERENCE TO LOCAL CONTEXT* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *VALUE*.
- 2) The following components of *MACH* are identified:
 - a) *ABC*, the alphabet, formed as the disjoint union of the following:
 - i) *SVV*, the set of names of node variables.
 - ii) *SEV*, the set of names of edge variables.
 - iii) *SPS*, the set of subpath symbols.
 - iv) *SAS*, the set of anonymous symbols.
 - v) *SBS*, the set of bracket symbols.

ISO/IEC 39075:2024(en)
22.6 Application of bindings to evaluate an expression

- b) *REDUCE*, the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.

- 3) Let *LC* be the subset of *MPB* that is indicated by *RTL*.

NOTE 400 — The local context is passed “by reference” in order to correctly evaluate non-local singletons. For example, given the pattern:

$((A) \rightarrow ((B) \rightarrow (C) \text{ WHERE } A.X = B.X + C.X) \rightarrow (D)) \{2\}$

then *A.X* makes a non-local reference to element variable *A*. A word of this pattern will repeat the outer <parenthesized path pattern expression> twice, requiring two evaluations of the WHERE clause. Each evaluation of the WHERE clause must locate the appropriate non-local reference to *A*. The bindings to the inner <parenthesized path pattern expression>, if passed “by value”, might not be enough information to determine the appropriate binding to the outer <parenthesized path pattern expression>.

- 4) Let *VREFS* be the binding variable references that are contained in *EXP* whose referenced binding variable is an element variable or a path variable declared by *GP* that is in the scope of *EXP*. Let *N* be the number elements of *VREFS*. For *i*, $1 \text{ (one)} \leq i \leq N$, let *VR_i* be the *i*-th element of *VREFS*. The *projected field value* is a value defined for the binding variable references in *VREFS* according to the following General Rules.
- 5) For each binding variable reference *BVR* in *VREFS*, the projected field value of *BVR* is defined as follows.

Case:

- a) If *BVR* is a binding variable reference whose variable is an element variable, then:

- i) Let *DEG* be the degree of reference of *BVR*.
- ii) Let *EV* be the element variable of *BVR*.
- iii) Case:

- 1) If *LC* is equal to *MPB*, then let *SPACE* be *MPB*.

NOTE 401 — That is, the search space is the entire multi-path binding. This case arises in two circumstances: 1) the evaluation of a <parenthesized path pattern where clause> in the outermost <parenthesized path pattern expression> of a selective <path pattern>; 2) the evaluation of a <graph pattern where clause>.

- 2) Otherwise, let *LCBI* be the bracket index of the first bracket symbol in *LC*. Let *LCPPPE* be the <parenthesized path pattern expression> contained in *GP* whose bracket index is *LCBI*. Let *PP* be the <path pattern> containing *LCPPPE*.

NOTE 402 — “Bracket index” is defined in Subclause 22.2, “Machinery for graph pattern matching”.

Case:

- A) If *EV* is not declared by *PP*, then let *SPACE* be *MPB*.

NOTE 403 — In this case, *EV* is declared in some other <path pattern> than the one that contains *BVR*. *BVR* is a non-local reference, therefore *DEG* is singleton, and *EV* is exposed as a singleton by the <path pattern>(s) that declares it.

- B) If *EV* is declared by *LCPPPE*, then let *SPACE* be *LC*.

NOTE 404 — In this case, *EV* is declared locally to *LCPPPE* and the binding(s) to *BVR* can be found by searching the local context *LC*.

- C) Otherwise, let *DEFPPPE* be the innermost <parenthesized path pattern expression> that declares *EV* and that contains *LCPPPE*. Let *BI* be the bracket

ISO/IEC 39075:2024(en)
22.6 Application of bindings to evaluate an expression

index of *DEFPPPE*. Let $[_{BI}$ and $]_{BI}$ be the bracket symbols whose bracket index is *BI*. Let *SPACE* be the smallest substring of *MPB* containing *LC* and beginning with $[_{BI}$ and ending with $]_{BI}$.

NOTE 405 — In this case, *EV* is declared in some outer scope containing *LCPPPE*, and the binding of *BVR*, if any, is found by searching the innermost scope that declares *EV*. *BVR* is a non-local reference, therefore *DEG* is singleton, and *EV* is exposed as a singleton by *DEFPPPE*.

NOTE 406 — “Bracket index” is defined in Subclause 22.2, “Machinery for graph pattern matching”.

iv) Case:

1) If *DEG* is singleton, then

Case:

A) If there is an elementary binding *EB* of *EV* in *SPACE*, then let *LOE* be a list with a single graph element, the graph element that is bound to *EV* by *EB*.

NOTE 407 — Even if *EV* is bound multiple times in *SPACE* (expressing an equijoin on *EV*), the list has only one graph element.

B) Otherwise, let *LOE* be the empty list.

NOTE 408 — This case can only arise if *DEG* is conditional singleton.

2) If *DEG* is group, then

Case:

A) If *SPACE* does not contain an elementary binding of *EV*, then let *LOE* be an empty list.

B) Otherwise, let *LOE* be the list of the graph elements that are bound to *EV* by elementary bindings in the order that they occur in *SPACE*, scanning *SPACE* from left to right, and retaining duplicates.

v) *LOE* is the list of graph elements bound to *BVR*.

vi) The projected field value of *EV* is defined as follows.

Case:

1) If *DEG* is singleton, then:

A) If *EV* is a node variable and *LOE* contains a single node *N*, then the projected field value of *BVR* is a node reference value for *N*.

B) If *EV* is an edge variable and *LOE* contains a single edge *E*, then the projected field value of *BVR* is an edge reference value for *E*.

C) Otherwise, *DEG* is conditional singleton, *LOE* is empty, and the projected field value of *BVR* is the null value.

2) Otherwise, *DEG* is group.

Case:

A) If *EV* is a node variable, then the projected field value of *BVR* is the list value comprising node reference values for the nodes given by *LOE* (in the order determined by *LOE*).

ISO/IEC 39075:2024(en)
22.6 Application of bindings to evaluate an expression

- B) If *EV* is an edge variable, then the projected field value of *BVR* is the list value comprising edge reference values for the edges given by *LOE* (in the order determined by *LOE*).
- b) If *BVR* is a binding variable reference whose variable is a path variable, then:
- i) Let *PV* be the path variable of *BVR*.
 - ii) Let *PB* be the path binding of *MPB* corresponding to the <path pattern> that declares *PV*.
 - iii) Let *EP* be the extracted path of *PB* and let *PEL* be the graph element list whose elements are graph reference values for the graph elements given by *EP* (in the order determined by *EP*).
 - iv) The projected field value of *BVR* is the path value whose path element list is *PEL*.
- 6) Let *R* be a record with fields *F_i*, $1 \text{ (one)} \leq i \leq N$, whose name is the variable name of *VR_i* and whose value is the projected field value of *VR_i*.
- 7) Let *V* be the result of evaluating *EXP* in a new child execution context amended with *R*.
- NOTE 409 — The type of the working record of this child execution context is determined by [Syntax Rule 25](#) of Subclause 16.3, "<graph pattern binding table>".
- 8) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *V* as *VALUE*.

Conformance Rules

None.

22.7 Evaluation of an expression on a group variable

Function

Evaluate a value expression that contains references to exactly one variable whose type is a group list value type.

Subclause Signature

“Evaluation of an expression on a group variable” [General Rules] (
Parameter: “GROUP LIST BINDING VARIABLE”,
Parameter: “EXPRESSION”
) Returns: “LIST VALUE”

GROUP LIST BINDING VARIABLE — a <binding variable> with group degree of reference

EXPRESSION — a <value expression> that references GROUP LIST BINDING VARIABLE

Syntax Rules

None.

General Rules

- 1) Let *GLBV* be the *GROUP LIST BINDING VARIABLE* and let *EXP* be the *EXPRESSION* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *LIST VALUE*.
- 2) Let *NAME* be the name of *GLBV*.
- 3) Let *CWR* be the current working record.
- 4) Let *ELTS* be the value of the field of *CWR* whose name is *NAME* and let *NELTS* be the number of elements of *ELTS*.

NOTE 410 — *ELTS* is always a list of graph element reference values.

- 5) Let *LV* be a list value determined as follows.
 - a) Initially, *LV* is an empty list.
 - b) For every *i*, $1 \leq i \leq NELTS$:
 - i) Let *FIELD* be the field whose name is *NAME* and whose value is the *i*-th element of *ELTS*.
 - ii) In a new child execution context whose working record is amended with the record comprising *FIELD*, the result of *EXP* is appended to *LV*.
- 6) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *LV* as *LIST VALUE*.

Conformance Rules

- 1) Without Feature GQ17, “Element-wise group variable operations”, in conforming GQL language, *EXP* shall be a <binding variable reference> to *GLBV*.

22.8 Application of bindings to generate a record

Function

Generate a record of output from a <graph pattern binding table>.

Subclause Signature

“Application of bindings to generate a record” [General Rules] (
Parameter: “GRAPH PATTERN”,
Parameter: “YIELD CLAUSE”,
Parameter: “MULTI-PATH BINDING”,
Parameter: “MACHINERY”
) Returns: “RECORD”

GRAPH PATTERN — a <graph pattern>.

YIELD CLAUSE — a <graph pattern yield clause>.

MULTI-PATH BINDING — a multi-path binding.

MACHINERY — the machinery for graph pattern matching

RECORD — a record to be output by <graph pattern binding table>.

Syntax Rules

None.

General Rules

- 1) Let *GP* be the *GRAPH PATTERN*, let *YC* be the *YIELD CLAUSE*, let *MPB* be the *MULTI-PATH BINDING*, and let *MACH* be the *MACHINERY* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *RECORD*.
- 2) Let *RTL* be reference to *MPB* as a subset of itself.
- 3) Let *N* be the number of <graph pattern variable>s simply contained in *YC*. For *i*, $1 \leq i \leq N$, let *YC_i* be the *i*-th <graph pattern variable> simply contained in *YC* and let *RECORD* be a record with *N* fields *F_i* determined as follows:
 - a) The name of *F_i* is *YC_i*.
 - b) The value of *F_i* is defined as follows:
 - i) The General Rules of Subclause 22.6, “Application of bindings to evaluate an expression”, are applied with *GP* as *GRAPH PATTERN*, *YC_i* as *EXPRESSION*, *MACH* as *MACHINERY*, *MPB* as *MULTI-PATH BINDING*, and *RTL* as *REFERENCE TO LOCAL CONTEXT*; let *VE* be the *VALUE* returned from the application of those General Rules.
 - ii) The value of *F_i* is *VE*.
- 4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *RECORD* as *RECORD*.

Conformance Rules

None.

IECNORM.COM : Click to view the full PDF of ISO/IEC 39075:2024

22.9 Resolution of a <simple directory path> from a start directory

Function

Determine a GQL-directory by resolving a <simple directory path> from a start directory

Subclause Signature

"Resolution of a <simple directory path> from a start directory" [Syntax Rules] (
Parameter: "SIMPLE DIRECTORY PATH",
Parameter: "START DIRECTORY"
) Returns: "RESOLVED DIRECTORY"

SIMPLE DIRECTORY PATH — a <simple directory path>.

START DIRECTORY — the GQL-directory from which SIMPLE DIRECTORY PATH is to be resolved.

RESOLVED DIRECTORY — the resolved GQL-directory.

Syntax Rules

- 1) Let *SDP* be the *SIMPLE DIRECTORY PATH* and let *SD* be the *START DIRECTORY* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *RESOLVED DIRECTORY*.
- 2) Let *DNSEQ* be the sequence of all <directory name>s that are immediately contained in *SDP*. Let *N* be the number of elements of *DNSEQ*.
- 3) Let *D*₀ be *SD* and for *i*, 1 (one) ≤ *i* ≤ *N*, let *DN*_{*i*} be the *i*-th element of *DNSEQ*.
- 4) For *j*, 1 (one) ≤ *j* ≤ *N*, let the GQL-directory *D*_{*j*} be defined as follows:
 - a) GQL-directory *D*_{*j-1*} shall contain a GQL-directory with name *DN*_{*j*}.
 - b) For every other GQL-directory *OGD* contained in *D*_{*j-1*}, if the name by which *OGD* is identified in *D*_{*j-1*} and *D*_{*j*} are visually confusable with each other, then an exception condition is raised: *syntax error or access rule violation — use of visually confusable identifiers (42004)*.
 - c) *D*_{*j*} is the GQL-directory with name *DN*_{*j*} contained in GQL-directory *D*_{*j-1*}.
- 5) Let *RD* be the GQL-directory *D*_{*N*}.
- 6) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *RD* as *RESOLVED DIRECTORY*.

General Rules

None.

Conformance Rules

None.

22.10 Store assignment

Function

Specify rules for store assignments.

NOTE 411 — See Subclause 4.17.3, “Assignment and store assignment”.

Subclause Signature

“Store assignment” [Syntax Rules] (
Parameter: “TARGET”,
Parameter: “VALUE”
)

TARGET — a site that is the target of an assignment operation.

VALUE — a <value expression> that is assigned to TARGET.

“Store assignment” [General Rules] (
Parameter: “TARGET”,
Parameter: “VALUE”
)

TARGET — a GQL site that is the target of an assignment operation.

VALUE — a GQL value that is assigned to TARGET.

Syntax Rules

- 1) Let *T* be the *TARGET* and let *V* be the *VALUE* in an application of the Syntax Rules of this Subclause.
- 2) Let *SD* be the declared type of *V*.
- 3) Let *TD* be the declared type of *T*.
- 4) *SD* shall be assignable to *TD*.
- 5) If at least one of *SD* or *TD* is a dynamic union type, then:
 - a) Let *SCTS* be the set defined as follows. If *SD* is a dynamic union type, then *SCTS* is the set of component types of *SD*; otherwise, *SCTS* is the singleton set comprising *SD*.
 - b) Let *TCTS* be the set defined as follows. If *TD* is a dynamic union type, then *TCTS* is the set of component types of *TD*; otherwise, *TCTS* is the singleton set comprising *TD*.
 - c) For each type *ST* in *SCTS*, there shall be one type *TT* in *TCTS* such that given a temporary site *TS* whose declared type is *TT* and an arbitrary expression *AX* whose declared type is *ST*, then the Syntax Rules of this Subclause are applied with *TS* as *TARGET* and *AX* as *VALUE*.
 - d) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.
- 6) If *TD* is a predefined value type, then

Case:

NOTE 412 — This ensures that every possible source value is assignable to the target using the General Rules of this Subclause.

- a) If *TD* is a reference value type, then:
 - i) The data type descriptors of *SD* and *TD* shall specify the same object base type name.
 - ii) Either *TD* is an open type or the constraining GQL-object type of *SD* shall be a subtype of the constraining GQL-object type of *TD*.
 - b) Otherwise, *TD* is not a reference value type.
- 7) If *TD* is a list value type, then:
- a) *SD* shall be a list value type.
 - b) Let *TELT* be the list element type of *TD* and let *VELT* be the list element type of *SD*.
 - c) Let *TS* be a transient site whose declared type is *TELT*, and let *AX* be an arbitrary expression whose declared type is *VELT*.
 - d) The Syntax Rules of this Subclause are applied with *TS* as *TARGET* and *AX* as *VALUE*.
- 8) If *TD* is a record type, then
- Case:
- a) If *TD* is an open record type, then *SD* shall be a record type.
 - b) Otherwise, *TD* is a closed record type.
- Case:
- i) If *SD* is a closed record type with the same set of field type names as *TD*. For every name *FTN* in the set of field type names of *SD*:
 - 1) Let *FTD* be the value type of the field type of *TD* whose name is *FTN*, let *FSD* be the value type of the field type of *SD* whose name is *FTN*, let *TS* be a transient site whose declared type is *FTD*, and let *AX* be an arbitrary expression whose declared type is *FSD*.
 - 2) The Syntax Rules of this Subclause are applied with *TS* as *TARGET* and *AX* as *VALUE*.
 - ii) Otherwise, *SD* is an open record type.
- 9) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

General Rules

- 1) Let *T* be the *TARGET* and let *V* be the *VALUE* in an application of the General Rules of this Subclause.
 - 2) Let *VD* be defined as follows.
- Case:
- a) If the declared type of *V* is a dynamic union type, then *VD* is the most specific static value type of *V*.
 - b) Otherwise, *VD* is the declared type of *V*.
- 3) Let *TD* be the declared type of *T*.
- 4) Case:
- a) If *V* is the null value, then *TD* shall be nullable and *T* is set to the null value.

b) Otherwise, V is material.

Case:

i) If TD is a dynamic union type, then:

- 1) Let $NTCTS$ be the set of component types of TD to which VD is assignable.
- 2) The Syntax Rules of Subclause 22.20, “Determination of value type precedence”, are applied with $NTCTS$ as $NDTSET$; let $NTCTL$ be the $NDTLIST$ returned from the application of those Syntax Rules.
- 3) Let TST be the value type of an element of $NTCTL$ for which a hypothetical application of the General Rules of this Subclause would succeed without raising an exception condition, or the first element of $NTCTL$ if such a hypothetical application would fail.
- 4) The General Rules of this Subclause are applied with a temporary site TS whose declared type is TST as $TARGET$ and V as $VALUE$.

ii) If TD is a reference value type, then:

- 1) If the base type of V is not the same as the base type of TD , then an exception condition is raised: *data exception — reference value, invalid base type (22G0V)*.
- 2) If the referent of V is not an object of TD , then an exception condition is raised: *data exception — reference value, invalid constrained type (22G0W)*.
- 3) The value of T is set to V .

iii) If TD is a fixed-length character string type with length in characters L , then:

Case:

- 1) If the length in characters M of V is greater than L , then

Case:

- A) If the rightmost $M-L$ characters of V are all <truncating whitespace> characters, then the value of T is set to the first L characters of V .
- B) Otherwise, one or more of the rightmost $M-L$ characters of V are not <truncating whitespace> characters and an exception condition is raised: *data exception — string data, right truncation (22001)*.

- 2) If 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>.
- 3) Otherwise, the length in characters of V is equal to L and the value of T is set to V .

iv) If TD is variable-length character string type with minimum length in characters $MINL$ and maximum length in characters $MAXL$, then

Case:

- 1) If the length in characters M of V is less than $MINL$, then T is set to V extended on the right by $MINL-M$ <space>s.
- 2) If the length in characters M of V is greater than $MAXL$, then

Case:

- A) If the rightmost $M-MAXL$ characters of V are all <truncating whitespace> characters, then the value of T is set to the first $MAXL$ characters of V .
 - B) Otherwise; one or more of the rightmost $M-MAXL$ characters of V are not <truncating whitespace> characters, then an exception condition is raised: *data exception — string data, right truncation (22001)*.
 - 3) Otherwise, the length in characters M of V is both greater than or equal to $MINL$ and less than or equal to $MAXL$. The value of T is set to V .
- v) If TD is a fixed-length byte string type with length in bytes L , then
- Case:
- 1) If the length in bytes M of V is greater than L , then
- Case:
- A) If the rightmost $M-L$ bytes of V are all equal to $X'00'$, then the value of T is set to the first L bytes of V .
 - B) Otherwise, one or more of the rightmost $M-L$ bytes of V are not equal to $X'00'$, and an exception condition is raised: *data exception — string data, right truncation (22001)*.
 - 2) If the length in bytes M of V is less than L , then the first M bytes of T are set to V and the last $L-M$ bytes of T are set to $X'00'$'s.
 - 3) Otherwise, the length in bytes of V is equal to L . The value of T is set to V .
- vi) If TD is a variable-length byte string type with minimum length in bytes $MINL$ and maximum length in bytes $MAXL$, then
- Case:
- 1) If the length in bytes M of V is less than $MINL$, then T is set to V extended on the right by $MINL-M$ $X'00'$'s.
 - 2) If the length in bytes M of V is greater than $MAXL$, then
- Case:
- A) If the rightmost $M-MAXL$ bytes of V are all equal to $X'00'$, then the value of T is set to the first $MAXL$ bytes of V .
 - B) Otherwise, one or more of the rightmost $M-MAXL$ bytes of V are not equal to $X'00'$, and an exception condition is raised: *data exception — string data, right truncation (22001)*.
 - 3) Otherwise, the length in bytes M of V is both greater than or equal to $MINL$ and less than or equal to $MAXL$. The value of T is set to V .
- vii) If TD is a numeric type, then
- Case:
- 1) If V is a value of TD , then the value of T is set to V .
 - 2) If a value of TD can be obtained from V by rounding or truncation, then it is implementation-defined (IA021) which of the following occurs:

- A) The value of *T* is set to that value. If *TD* is exact numeric, then it is implementation-defined (IA005) whether the approximation is obtained by rounding or by truncation.
 - B) An exception condition is raised: *data exception — numeric value out of range* (22003).
 - 3) Otherwise, an exception condition is raised: *data exception — numeric value out of range* (22003).
- viii) If *TD* is a datetime type or a time type, then
 - 1) If only one of *TD* and *VD* is a zoned datetime type or a zoned time type, then *V* is effectively replaced by
`CAST ( V AS TD )`
 - 2) Case:
 - A) If *V* is a value of *TD*, then the value of *T* is set to *V*.
 - B) If a value of *TD* can be obtained from *V* by rounding or truncation, then it is implementation-defined (IA021) which of the following occurs:
 - I) The value of *T* is set to that value. It is implementation-defined (IA005) whether the approximation is obtained by rounding or truncation.
 - II) An exception condition is raised: *data exception — datetime field overflow* (22008).
 - C) Otherwise, an exception condition is raised: *data exception — datetime field overflow* (22008).
- ix) If *TD* is a list value type, then
 - Case:
 - 1) If the maximum cardinality *L* of *T* is equal to the cardinality of *V*, then for *i* ranging from 1 (one) to *L*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*.
 - 2) If the maximum cardinality *L* of *T* is smaller than the cardinality *M* of *V*, then
 - Case:
 - A) If the rightmost *M*–*L* elements of *V* are all null, then for *i* ranging from 1 (one) to *L*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*.
 - B) If one or more of the rightmost *M*–*L* elements of *V* are not the null value, then an exception condition is raised: *data exception — array data, right truncation* (2202F).
 - 3) If the maximum cardinality *L* of *T* is greater than the cardinality *M* of *V*, then for *i* ranging from 1 (one) to *M*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*. The cardinality of the value of *T* is set to *M*.

NOTE 413 — The maximum cardinality *L* of *T* is unchanged.

- x) If *TD* is a closed record type, then:

- 1) If there is a field of V whose name is not the name of a field type of TD , then an exception condition is raised: *data exception — record data, field unassignable (22G0X)*.
 - 2) If there is a field type of TD whose name is not the name of a field of V , then an exception condition is raised: *data exception — record data, field missing (22G0Y)*.
 - 3) For every name FTN in the set of field type names of VD , the General Rules of this Subclause are applied with the field with name FTN of T as *TARGET* and the field with name FTN of V as *VALUE*.
- xi) If TD is an open record type, then for every field F in V :
- 1) Let FTN be the name of F , let FTD be the declared type of F , and let TS be a transient site whose declared type is FTD .
 - 2) The General Rules of this Subclause are applied with TS as *TARGET* and F as *VALUE*.
 - 3) The field with name FTN in T is set to the value of TS .
- xii) Otherwise, the value of T is set to V .
- 5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

22.11 Determination of identical values

Function

Determine whether two values are identical.

Subclause Signature

“Determination of identical values” [General Rules] (
Parameter: “FIRST VALUE”,
Parameter: “SECOND VALUE”
)

FIRST VALUE — the first of two values for which it is to be determined whether they are identical values.

SECOND VALUE — the second of two values for which it is to be determined whether they are identical values.

Syntax Rules

None.

General Rules

- 1) Let *V1* be the *FIRST VALUE* and let *V2* be the *SECOND VALUE* in an application of the General Rules of this Subclause.

NOTE 414 — This Subclause is invoked implicitly wherever the word “identical” is used of two values.

- 2) Whether *V1* is identical to *V2* is determined as follows.

Case:

- a) If *V1* and *V2* are both the null value, then *V1* is identical to *V2*.
- b) If *V1* is the null value and *V2* is not the null value, or if *V1* is not the null value and *V2* is the null value, then *V1* is not identical to *V2*.
- c) If *V1* and *V2* are character strings, then let *L* be `CHARACTER_LENGTH(V1)` and

Case:

- i) If `CHARACTER_LENGTH(V2)` equals *L*, and if for all *i*, $1 \text{ (one)} \leq i \leq L$, the *i*-th character of *V1* corresponds to the same Unicode code point as the *i*-th character of *V2*, then *V1* is identical to *V2*.
 - ii) Otherwise, *V1* is not identical to *V2*.
- d) If *V1* and *V2* are both numbers, then

Case:

- i) If *V1* and *V2* are both exact numbers, then *V1* is identical to *V2* if and only if *V1* is not distinct from *V2*.

ISO/IEC 39075:2024(en)
22.11 Determination of identical values

- ii) If *V1* and *V2* are both approximate numbers, then *V1* is identical to *V2* if and only if *V1* is not distinct from *V2*.

- iii) Otherwise, *V1* and *V2* are not identical.

NOTE 415 — Numbers are always compared by their numeric value, as specified in Subclause 19.3, “<comparison predicate>”, and therefore two numbers with the same numeric value are generally indistinguishable unless an explicit <value type predicate> is used. See Subclause 19.6, “<value type predicate>”.

NOTE 416 — The distinction between exact numbers and approximate numbers made by this General Rule only impacts GQL-implementations that support dynamic union types where it enables the correct functioning of any dynamically generated type tests and casts. See Subclause 4.14.4, “Dynamic generation of type tests and casts”.

- e) If *V1* and *V2* are zoned times or zoned datetimes, then:

Case:

- i) If *V1* and *V2* are not distinct and their time zone displacement fields are not distinct, then *V1* is identical to *V2*.
- ii) Otherwise, *V1* is not identical to *V2*.

- f) If *V1* and *V2* are records, then:

Case:

- i) If the sets of field types of *V1* and *V2* have the same cardinality and all respective fields of *V1* and *V2* with the same name are identical, then *V1* is identical to *V2*.
- ii) Otherwise, *V1* is not identical to *V2*.

- g) If *V1* and *V2* are lists, then:

Case:

- i) If *V1* and *V2* have the same cardinality and elements in the same ordinal position in the two lists are identical, then *V1* is identical to *V2*.
- ii) Otherwise, *V1* is not identical to *V2*.

- h) Otherwise,

Case:

- i) If *V1* and *V2* are comparable values, then *V1* is identical to *V2* if and only if *V1* is not distinct from *V2*.
- ii) Otherwise, *V1* and *V2* are not identical.