



PUBLIC

SAP HANA Platform 2.0 SPS 04

Document Version: 1.1 – 2019-10-31

SAP HANA Graph Reference

Content

- 1 SAP HANA Graph Reference. 4**
- 2 Introduction. 5**
- 3 SAP HANA Graph Data Model. 6**
- 4 Graph Workspaces. 9**
 - 4.1 Create and Drop Graph Workspaces. 9
 - 4.2 Export and Import Graph Workspaces. 11
- 5 Graph Data Modification. 12**
- 6 GraphScript Language. 14**
 - 6.1 Data Types. 14
 - 6.2 General Script Structure. 17
 - 6.3 Integration of GraphScript into Stored Procedure Environment. 18
 - 6.4 Types. 18
 - 6.5 Comments. 18
 - 6.6 Expressions. 19
 - 6.7 Statements. 32
 - 6.8 Built-In Functions. 38
 - 6.9 Reserved Keywords. 42
 - 6.10 Restrictions for GraphScript Procedures. 48
 - 6.11 Complex GraphScript Examples. 48
- 7 openCypher Pattern Matching. 50**
 - 7.1 OPENCYPHER_TABLE SQL Function. 50
 - 7.2 openCypher Query Language. 52
 - Match Clause. 52
 - Return Clause. 56
 - Keywords. 59
 - Built-In Functions. 60
 - Basic Building Blocks. 63
- 8 Graph Algorithms. 65**
 - 8.1 Neighborhood Search (Breadth-First Search). 65
 - 8.2 Shortest Path. 67
 - Shortest Path (One-to-All). 67
 - Shortest Path (One-to-One). 71

8.3	Strongly Connected Components.	73
8.4	Graph Algorithm Variables.	75
8.5	Pattern Matching.	76
	Graphical Pattern Editor.	76
	Query Language.	77
9	Additional Information.	79
9.1	Appendix A – SAP HANA Graph Viewer.	79
	Install SAP HANA Graph Viewer.	79
9.2	Appendix B - Greek Mythology Graph Example.	80
9.3	Appendix C - Notation.	82

1 SAP HANA Graph Reference

This reference provides information about SAP HANA Graph. It is organized as follows:

- Introduction
Introduction to SAP HANA Graph.
- SAP HANA Graph Data Model
Description of SAP HANA Graph data model using a simple example.
- Graph Workspaces
Description of SQL statements for creating and manipulating graph workspaces.
- Graph Data Modification
Description of SQL statements for modifying graph data.
- GraphScript Language
Description of the GraphScript stored procedure language.
- openCypher Query Language
Description of the openCypher query language interface in SAP HANA (Cypher is a registered trademark of Neo4j, Inc.).
- Graph Algorithms
Description of supported graph algorithms in SAP HANA calculation scenarios.
- Additional Information
A collection of additional information.

2 Introduction

SAP HANA Graph is an integral part of SAP HANA core functionality. It expands the SAP HANA platform with native support for graph processing and allows you to execute typical graph operations on the data stored in an SAP HANA system.

Caution

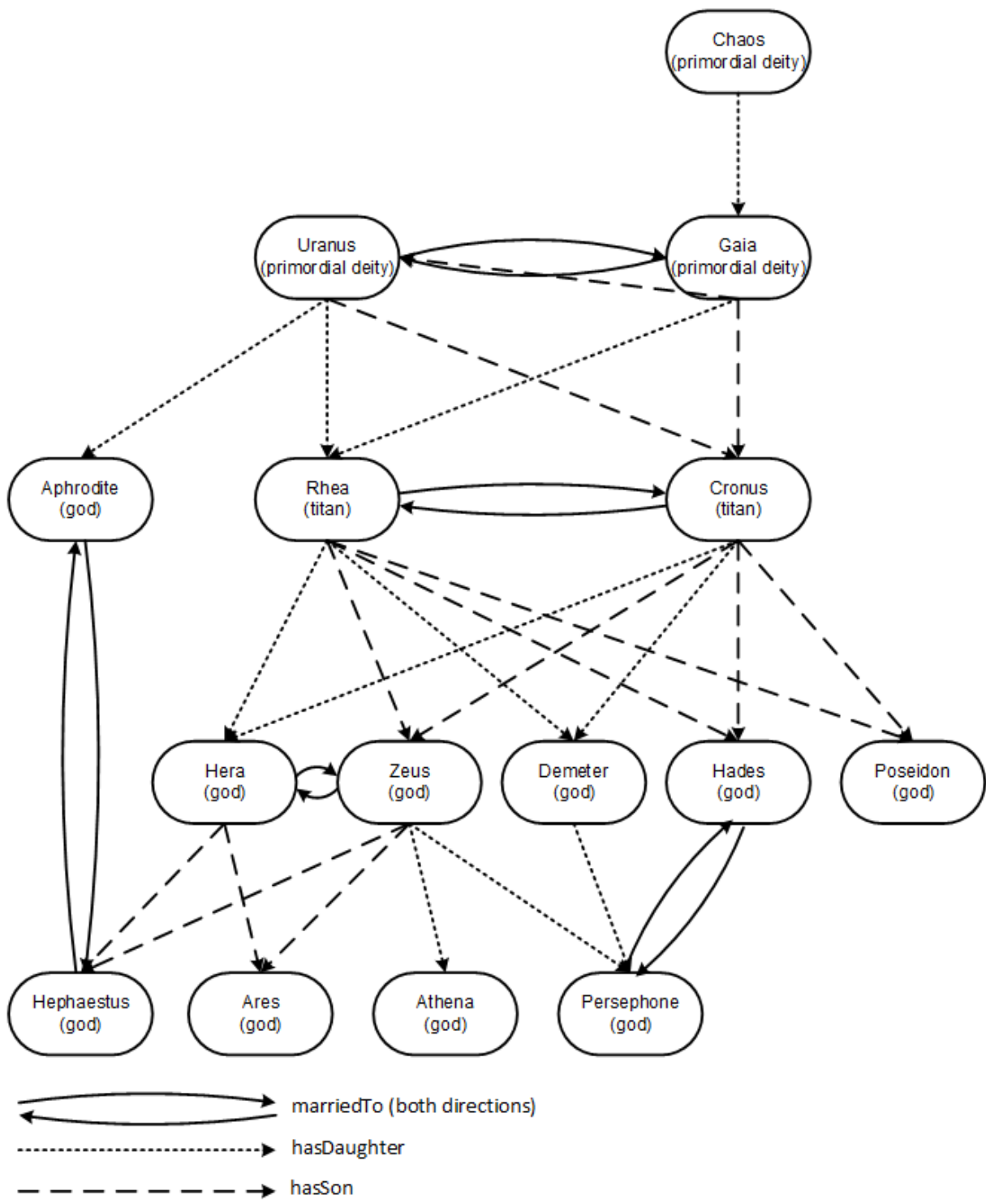
The usage of cross-database access in a scenario with more than one tenant database in combination with SAP HANA Graph is not supported.

3 SAP HANA Graph Data Model

Graphs are a powerful abstraction that can be used to model different kinds of networks and linked data coming from many industries, such as logistics and transportation, utility networks, knowledge representation, text processing, and many more.

In SAP HANA, a graph is a set of vertices and a set of edges. Each edge connects two vertices; one vertex is denoted as the source and the other as the target. Edges are always directed. Any number of edges may connect the same two vertices. Vertices and edges can have an arbitrary number of attributes. Such an attribute consists of a name that is associated with a data type and a value.

The following image provides an example of a graph in which vertices represent Greek mythology members and edges represent the relationships among them. All vertices have attributes "NAME" (shown in the image) and "TYPE" (shown in the image). "TYPE" takes one of the following values: 'primordial deity', 'god', 'titan'. Some vertices have an attribute "RESIDENCE" (not shown in the image). All edges have attributes "KEY" (not shown in the image) and "TYPE", which takes one of the following values: 'marriedTo', 'hasSon', and 'hasDaughter'.



The primary storage of a graph are two relational objects that can be tables or views or table or view synonyms. We will refer to them as vertex table and edge table for the sake of simplicity. The vertex table stores the set of

vertices and the edge table stores the set of edges. Vertex attributes match to columns of the vertex table. Edge attributes match to columns of the edge table. The maximum number of attributes is bound by the maximum number of columns for the underlying tables (for more information see the SAP HANA SQL Reference Guide). One of the vertex attributes must uniquely identify vertices. This attribute is also referred to as vertex key. Similarly, one of the edge attributes must uniquely identify edges and is referred to as edge key. The edge table contains two additional columns referencing the key column of the vertex table. One of them identifies the source vertex and the other identifies the target vertex of an edge.

The following tables show the tabular storage of the Greek mythology graph.

NAME (Unique Key)	TYPE	RESIDENCE
Cronus	titan	Tartarus
Rhea	titan	Tartarus
Zeus	god	Olympus
Hades	god	Underworld
...

KEY (Unique Key)	SOURCE	TARGET	TYPE
1	Cronus	Rhea	marriedTo
2	Rhea	Cronus	marriedTo
3	Cronus	Zeus	hasSon
4	Rhea	Zeus	hasSon
...

Relational storage allows the whole feature set of SAP HANA to be applied to the graph data: access control, backup and recovery, etc. It also allows all SAP HANA Graph functions to be applied to the graph data stored in relational format coming from business applications. SAP HANA Graph provides a dedicated catalog object, which is referred to as a graph workspace, for defining a graph in terms of the existing SAP HANA tables.

Related Information

[SAP HANA SQL and System Views Reference](#)

4 Graph Workspaces

A graph workspace is a catalog object that defines a graph in terms of tables and columns:

- Vertex table
- Edge table
- Key column in the vertex table
- Key column in the edge table
- Source vertex column in the edge table
- Target vertex column in the edge table

⚠ Caution

Caching is not supported for graph workspaces that refer to virtual tables.

A graph workspace is uniquely identified by the database schema it resides in and the workspace name. An SAP HANA instance can contain multiple graph workspaces in the same schema (with different workspace names) or different database schemas.

Graph workspace information is provided using the GRAPH_WORKSPACES system view.

Related Information

[SAP HANA SQL and System Views Reference](#)

4.1 Create and Drop Graph Workspaces

SAP HANA Graph provides SQL extensions for creating and dropping graph workspaces.

Before creating a graph workspace, check that the underlying tables, views or synonyms for vertices and edges exist. The following SQL commands create a vertex table "MEMBERS" and an edge table "RELATIONSHIPS" in a schema "GREEK_MYTHOLOGY".

```
CREATE SCHEMA "GREEK_MYTHOLOGY";
CREATE COLUMN TABLE "GREEK_MYTHOLOGY"."MEMBERS" (
  "NAME" VARCHAR(100) PRIMARY KEY,
  "TYPE" VARCHAR(100),
  "RESIDENCE" VARCHAR(100)
);
CREATE COLUMN TABLE "GREEK_MYTHOLOGY"."RELATIONSHIPS" (
  "KEY" INT UNIQUE NOT NULL,
  "SOURCE" VARCHAR(100) NOT NULL
  REFERENCES "GREEK_MYTHOLOGY"."MEMBERS" ("NAME")
  ON UPDATE CASCADE ON DELETE CASCADE,
  "TARGET" VARCHAR(100) NOT NULL
  REFERENCES "GREEK_MYTHOLOGY"."MEMBERS" ("NAME")
);
```

```
ON UPDATE CASCADE ON DELETE CASCADE,  
"TYPE" VARCHAR(100)  
);
```

Having created all necessary tables, we can now create a graph workspace "GRAPH" in the schema "GREEK_MYTHOLOGY" with the following CREATE GRAPH WORKSPACE statement.

```
CREATE GRAPH WORKSPACE "GREEK_MYTHOLOGY"."GRAPH"  
EDGE TABLE "GREEK_MYTHOLOGY"."RELATIONSHIPS"  
SOURCE COLUMN "SOURCE"  
TARGET COLUMN "TARGET"  
KEY COLUMN "KEY"  
VERTEX TABLE "GREEK_MYTHOLOGY"."MEMBERS"  
KEY COLUMN "NAME";
```

The vertex table, the edge table, and the graph workspace can reside in different schemas. If any of the schemas are omitted, the default schema is assumed.

The columns of the edge table and the vertex table are interpreted correspondingly as edge and vertex attributes. In the example in [SAP HANA Graph Data Model \[page 6\]](#), the vertices in the table "GREEK_MYTHOLOGY"."MEMBERS" have the attributes "TYPE" and "RESIDENCE", which may contain NULL values. A row in the vertex table or the edge table is interpreted correspondingly as a vertex or an edge.

The vertex and edge key columns can be of one of the following SQL types: TINYINT, SMALLINT, INTEGER, BIGINT, VARCHAR, and NVARCHAR. Integer types (TINYINT, SMALLINT, INTEGER, BIGINT) are recommended.

During the creation of a new graph workspace, the SAP HANA system checks that all specified tables and columns exist, and have supported data types. A newly created graph workspace is valid as long as the specified columns and tables exist and fulfill the following validity requirements.

Characteristics of vertex key and edge key columns:

- supported key types: TINYINT, SMALLINT, INTEGER, BIGINT, VARCHAR, and NVARCHAR
- NOT NULL flag
- UNIQUE flag

Characteristics of source and target columns:

- same data type as the vertex key column
- NOT NULL flag

A valid graph workspace is consistent if both source and target columns contain existing values from the vertex key column. In the CREATE statements listed above, the referential constraints ("REFERENCES") are used to guarantee the consistency of the workspace.

An existing graph workspace "GRAPH" in schema "GREEK_MYTHOLOGY" can be deleted with the following statement.

```
DROP GRAPH WORKSPACE "GREEK_MYTHOLOGY"."GRAPH";
```

Creating or dropping graph workspaces does not modify the content of the underlying vertex and edge tables.

To create a graph workspace, a user needs the CREATE ANY privilege for the intended schema and the SELECT privilege for both vertex and edge tables. To drop a graph workspace, a user must be the creator of the graph workspace or must have the DROP privilege for the given graph workspace.

Related Information

[SAP HANA Graph Data Model \[page 6\]](#)

4.2 Export and Import Graph Workspaces

Graph workspace objects can be exported from or imported into an SAP HANA system using the existing export and import SQL commands.

Exporting or importing a schema automatically exports or imports all graph workspaces contained in that schema. For the full set of options for IMPORT and EXPORT commands, see the SAP HANA SQL Reference Guide . The vertex and edge tables are by default exported together with a graph workspace unless the NO DEPENDENCIES parameter is used.

Assuming the database instance is installed under `/usr/sap/HDB/HDB00` , the following statement exports the graph workspace "GREEK_MYTHOLOGY"."GRAPH" together with the vertex table "GREEK_MYTHOLOGY"."MEMBERS" and the edge table "GREEK_MYTHOLOGY"."RELATIONSHIPS" to `/usr/sap/HDB/HDB00/work`

```
EXPORT "GREEK_MYTHOLOGY"."GRAPH" AS BINARY INTO '/usr/sap/HDB/HDB00/work';
```

The following statement imports the graph workspace "GREEK_MYTHOLOGY"."GRAPH" together with the vertex table "GREEK_MYTHOLOGY"."MEMBERS" and the edge table "GREEK_MYTHOLOGY"."RELATIONSHIPS" from `/usr/sap/HDB/HDB00/work`

```
IMPORT "GREEK_MYTHOLOGY"."GRAPH" AS BINARY FROM '/usr/sap/HDB/HDB00/work';
```

Related Information

[SAP HANA SQL and System Views Reference](#)

5 Graph Data Modification

Any change to the edge table or the vertex table will affect the edges or the vertices of the graph.

The following SQL statements create two new vertices with vertex keys 'Oceanus' and 'Tethys' and an edge in the workspace "GREEK_MYTHOLOGY"."GRAPH". For the complete list of statements for creating the example graph in [SAP HANA Graph Data Model \[page 6\]](#), see [Appendix B - Greek Mythology Graph Example \[page 80\]](#).

```
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS" ("NAME", "TYPE", "RESIDENCE")
VALUES ('Oceanus', 'titan', 'Othrys');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS" ("NAME", "TYPE", "RESIDENCE")
VALUES ('Tethys', 'titan', 'Othrys');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS" ("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (35, 'Oceanus', 'Tethys', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS" ("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (36, 'Tethys', 'Oceanus', 'marriedTo');
```

Vertices can be modified and deleted using SQL update and delete statements. The following statement modifies the "RESIDENCE" attribute of all vertices with "RESIDENCE" 'Othrys'.

```
UPDATE "GREEK_MYTHOLOGY"."MEMBERS" SET "RESIDENCE" = 'Tartarus'
WHERE "RESIDENCE" = 'Othrys';
```

The following statement deletes those edges from the graph workspace "GREEK_MYTHOLOGY"."GRAPH" that contain 'Oceanus' as "SOURCE" or "TARGET".

```
DELETE FROM "GREEK_MYTHOLOGY"."RELATIONSHIPS"
WHERE "SOURCE" = 'Oceanus' or "TARGET" = 'Oceanus';
```

Vertices can be inserted, updated, and deleted in the same way as edges.

```
DELETE FROM "GREEK_MYTHOLOGY"."MEMBERS"
WHERE "NAME" IN ('Oceanus', 'Tethys');
```

If the consistency of a graph workspace is not guaranteed by referential constraints (see [Create and Drop Graph Workspaces \[page 9\]](#)), modifications of the referenced graph tables can lead to an inconsistent graph workspace. The exact behavior in case of inconsistent graph workspaces is different for each Graph Engine component. Details are provided in the relevant chapters.

The uniqueness of vertices in the workspace "GREEK_MYTHOLOGY"."GRAPH" can be checked with the following query on the vertex table:

```
SELECT "NAME", COUNT(*)
FROM "GREEK_MYTHOLOGY"."MEMBERS"
GROUP BY "NAME"
HAVING COUNT(*) > 1;
```

If the result of above query is not empty, then the vertex key column ("NAME") contains duplicates and the graph workspace is inconsistent.

A similar query can be used on the edge table to check the uniqueness of edge keys:

```
SELECT "KEY", COUNT(*)
FROM "GREEK_MYTHOLOGY"."RELATIONSHIPS"
```

```
GROUP BY "KEY"  
HAVING COUNT(*) > 1;
```

The correctness of the source and target vertex references can be checked with the following query:

```
SELECT "KEY", "SOURCE", "TARGET"  
FROM "GREEK_MYTHOLOGY"."RELATIONSHIPS"  
WHERE  
    "SOURCE" NOT IN (SELECT "NAME" FROM "GREEK_MYTHOLOGY"."MEMBERS")  
    OR  
    "TARGET" NOT IN (SELECT "NAME" FROM "GREEK_MYTHOLOGY"."MEMBERS");
```

If the result of the above query is not empty, then either the source or target of the resulting edge is not a valid vertex key, leading to an inconsistent graph workspace.

6 GraphScript Language

GraphScript is an imperative programming language that provides application developers with a high-level interface for accessing the graph data defined by a graph workspace.

The GraphScript language is designed to ease the development of application-specific graph algorithms and to integrate them into SQL-based data processing. Furthermore, GraphScript provides optimized built-in algorithms to solve common graph-related problems, such as finding the shortest path from one vertex to another. These algorithms are available as built-in functions which can be re-used in any GraphScript program, thus greatly simplifying the development of efficient solutions for customer-specific graph-related problems.

In case of inconsistent graph workspaces, GraphScript procedures exit with an error.

GraphScript procedures support debugging from the SAP HANA Database Explorer. For more information see *Debug Procedures in the SAP HANA Database Explorer* in [SAP HANA Developer Guide for SAP HANA XS Advanced Model](#).

Related Information

[SAP HANA Developer Guide for XS Advanced Model](#)

6.1 Data Types

This section describes the data types that are available in GraphScript.

The following table summarizes the supported data types in GraphScript and classifies them by their characteristics as follows:

Classification	Data Type
Numeric types	INTEGER, BIGINT, DOUBLE
Boolean type	BOOLEAN
Character string types	VARCHAR, NVARCHAR
Graph-specific types	GRAPH, VERTEX, EDGE, WEIGHTEDPATH
Collection types	MULTISET, SEQUENCE
Datetime type	TIMESTAMP

Numeric Types

Each numeric type has a maximum value and a minimum value. A numeric overflow exception is thrown if a given value is smaller than the minimum allowed value or greater than the maximum allowed value.

- **INTEGER**
The INTEGER data type specifies a 32-bit signed integer. The minimum value is -2,147,483,648. The maximum value is 2,147,483,647. Integer literals that are not suffixed with an 'L' are treated as 32-bit integer literals. A variable of type INTEGER is default-initialized with value 0.
- **BIGINT**
The BIGINT data type specifies a 64-bit signed integer. The minimum value is -9,223,372,036,854,775,808. The maximum value is 9,223,372,036,854,775,807. Literal values of type BIGINT have to be suffixed with an 'L'. A variable of type BIGINT is default-initialized with value the 0L.
- **DOUBLE**
The DOUBLE data type specifies a double-precision 64-bit floating-point number. The minimum value is -1.7976931348623157E308 and the maximum value is 1.7976931348623157E308. The smallest positive DOUBLE value is 2.2250738585072014E-308 and the largest negative DOUBLE value is -2.2250738585072014E-308. A variable of type DOUBLE is default-initialized with value 0.0.

Boolean Type

The BOOLEAN data type stores Boolean values, which are TRUE and FALSE. A variable of type BOOLEAN is default-initialized with value FALSE.

Character String Types

The character string data types specify character strings. The VARCHAR data type specifies ASCII character strings and NVARCHAR is used for storing Unicode character strings.

- **VARCHAR**
The VARCHAR data type specifies a variable-length character string. We recommend using VARCHAR with ASCII character-based strings only.
- **NVARCHAR**
The NVARCHAR data type specifies a variable-length character string.

Datetime Type

- **TIMESTAMP**
The TIMESTAMP data type consists of date and time information. Its default format is 'YYYY-MM-DD HH24:MI:SS.FF7'. FF_n represents the fractional seconds where n indicates the number of digits in the fractional part. The range of the time stamp value is between 0001-01-01 00:00:00.0000000 and 9999-12-31 23:59:59.9999999. A variable of type TIMESTAMP has no default value and therefore must be initialized explicitly.

Graph-Specific Types

GraphScript supports four graph-specific data types: VERTEX, EDGE, GRAPH, and WEIGHTEDPATH.

- VERTEX
The VERTEX data type specifies a vertex (a node) in the graph. An existing vertex object can be retrieved from a graph workspace using its key. Supported key types are INTEGER, BIGINT, and VARCHAR. The creation of new vertices is not supported. A variable of type VERTEX must be initialized at the time of declaration. There is no default initialization for variables of type VERTEX.
- EDGE
The EDGE data type specifies an edge (a relationship) in the graph. An existing edge object can be retrieved from a graph workspace using its key. Supported key types are INTEGER, BIGINT, and VARCHAR. The creation of new edges is not supported. A variable of type EDGE must be initialized at the time of declaration. There is no default initialization for variables of type EDGE.
- WEIGHTEDPATH
The WEIGHTEDPATH data type specifies a path with an associated weight. A WEIGHTEDPATH consists of a sequence of edges or a single vertex (minimal path). The sequence of edges may also be empty. A WEIGHTEDPATH can be created by calling the built-in function SHORTEST_PATH with a graph and a pair of vertices as parameters. GraphScript supports several operations on WEIGHTEDPATH variables, such as extracting its vertices and edges as well as calculating the length and the weight of a path. A variable of type WEIGHTEDPATH must be initialized at the time of declaration. There is no default initialization for variables of type WEIGHTEDPATH. The WEIGHTEDPATH type needs to be specialized with a weight type. The following weight types are currently supported: INT, DOUBLE, BIGINT. These lead to the following WEIGHTEDPATH types: WEIGHTEDPATH<INT>, WEIGHTEDPATH<DOUBLE>, WEIGHTEDPATH<BIGINT>.
- GRAPH
A variable of type GRAPH specifies a graph. A graph can refer to a graph workspace catalog object specified by workspace name, an inverse graph for an already defined graph, or a subgraph of an already defined graph. Alternatively, a graph can be created from two input tables containing vertices and edges accordingly. A variable of type GRAPH must be initialized at the time of declaration. There is no default initialization for variables of type GRAPH.

Collection Types

A collection is a composite value comprising zero or more elements of the same type. Containers of vertices or edges can only take elements from a single graph. Container variables cannot be reassigned to a container with elements from another graph.

- MULTISSET
A multiset is an unordered collection. Since a multiset is unordered, there is no ordinal position to reference individual elements of a multiset. The elements in a MULTISSET can be of data type VERTEX, EDGE, INTEGER, BIGINT, DOUBLE, TIMESTAMP, VARCHAR, NVARCHAR or BOOLEAN. Furthermore, the elements of a MULTISSET can also be of type MULTISSET of VERTEX. Such a collection of collections is used as the return type of the STRONGLY_CONNECTED_COMPONENTS algorithm. Currently, multisets of multisets are very limited in functionality compared to multisets of non-collection data types.
- SEQUENCE
A sequence is a mutable, ordered collection. It is possible to access and modify any element via its position, which lies between 1 and the size of the sequence. A new sequence can be constructed from the concatenation of two sequences or the concatenation of a sequence and a single element. The elements in

a SEQUENCE can be of data type VERTEX, EDGE, INTEGER, BIGINT, DOUBLE, TIMESTAMP, VARCHAR, NVARCHAR or BOOLEAN.

Data Type Conversions

GraphScript does not support implicit conversions between objects of different data types. Numeric and character string expressions can be explicitly cast using the cast functions provided in the section [Built-In Functions \[page 38\]](#).

Typed Literals

A literal is a symbol that represents a specific fixed data value.

Character string literals

A character string literal is enclosed in single quotation marks. The character string literal is prefixed by an N if it is of type NVARCHAR.

Examples

```
'Brian' -- VARCHAR
'23' -- VARCHAR
N'Simon' -- NVARCHAR
```

Numeric literals

A numeric literal is represented by a sequence of numbers that are not enclosed in quotation marks. Numbers may contain a decimal point. BIGINT literals need to be suffixed with an L.

Examples

```
123 -- INT
123L -- BIGINT
123.4 -- DOUBLE
```

Boolean literals

A Boolean literal can be either true or false. Both literals are keywords in GraphScript and therefore case-insensitive.

6.2 General Script Structure

A GraphScript program consists of an arbitrary number of statements.

```
<script> ::= { <statement> }
```

6.3 Integration of GraphScript into Stored Procedure Environment

GraphScript can be used as SAP HANA stored procedure language. The mandatory language identifier for GraphScript is GRAPH.

```
CREATE PROCEDURE myGraphProc()
LANGUAGE GRAPH READS SQL DATA AS
BEGIN
    -- here goes the GraphScript program
END
```

6.4 Types

The type of an object is used in a definition statement to introduce new objects with the given type. Once an object has been declared, the type of the object is immutable.

```
<primitive_type> ::=
| <numeric_type>
| VARCHAR
| NVARCHAR
| BOOLEAN
| TIMESTAMP
<numeric_type> ::=
| INTEGER
| INT
| BIGINT
| DOUBLE
<vertex_or_edge> ::= VERTEX | EDGE
<weightedpath> ::= WEIGHTEDPATH <lower> <numeric_type> <greater>
<graph> ::= GRAPH
<multiset_type> ::=
    MULTISSET <lower> <vertex_or_edge> <greater>
    | MULTISSET <lower> <primitive_type> <greater>
    | MULTISSET <lower> MULTISSET <lower> VERTEX <greater> <greater>
<sequence_type> ::=SEQUENCE <lower> <vertex_or_edge> <greater> |
    SEQUENCE <lower> <primitive_type> <greater> |
```

6.5 Comments

GraphScript supports single-line comments as well as multi-line comments.

A single-line comment starts with "--"; all characters that follow in the line are treated as a comment. A multi-line comment is enclosed in "/*" and "*/".

```
Int v = 23; -- this is a single line comment
```

```
FOREACH e IN Edges(:g) {
```

```
/* this is a multi line comment
   Int i = :e.attr;
   Int v = 23;*/
}
```

6.6 Expressions

An expression is a language construct that returns a value of a given type.

GraphScript supports the following expressions:

- Literal expressions (`literal_expr`)
- Attribute access expressions (`attr_access_expr`)
- Local variable expressions (`local_var_expr`)
- Arithmetic expressions (`arithmetic_expr`)
- Relational expressions (`relational_expr`)
- Logical expressions (`logical_expr`)
- Collection initializer list expressions (`collection_init_expr`)
- Function expressions (`function_expr`)
- Set operations (`set_operation`)
- Concatenation expressions (`concat_expr`)
- Positional access expressions (`index_expr`)
- Cell access expressions (`cell_expr`)
- Filter expressions (`filter_expr`)
- Projection expressions (`proj_expr`)
- Closure expressions (`clos_expr`)

```
<expr> ::=
<literal_expr>
| <attr_access_expr>
| <local_var_expr>
| <arithmetic_expr>
| <relational_expr>
| <logical_expr>
| <collection_init_expr>
| <function_expr>
| <set_operation>
| <concat_expr>
| <index_expr>
| <cell_expr>
| <filter_expr>
| <proj_expr>
| <clos_expr>
```

Literal Expressions

A literal expression is a symbol that represents a specific fixed data value.

```
<int_literal> ::= 0 | <pos_digit> {<digit>}
<bigint_literal> ::= <int_literal> L
<double_literal> ::= ( 0 | <pos_digit> {<digit>}) <dot> <digit> {<digit>}
<numeric_literal> ::= <int_literal> | <bigint_literal> | <double_literal>
<varchar_literal> ::=
  <single_quote> {<any_character>-<single_quote>} <single_quote>
<nvarchar_literal> ::=
  N<single_quote> {<any_character>-<single_quote>} <single_quote>
<literal_exp> ::=
  <numeric_literal> | <varchar_literal> | <nvarchar_literal>
```

Examples

```
23
23L
23.5
'Brian'
N'Brian'
```

BNF Lowest Terms Representations

The following list summarizes special characters and symbols that are used in GraphScript.

```
<comma> ::= ,
<single_quote> ::= '
<double_quote> ::= "
<l_paren> ::= (
<r_paren> ::= )
<l_curly> ::= {
<r_curly> ::= }
<l_square> ::= [
<r_square> ::= ]
<pipe> ::= |
<semicolon> ::= ;
<colon> ::= :
<dot> ::= .
<equals> ::= =
<minus> ::= -
<plus> ::= +
<lower> ::= <
<lower_equal> ::= <=
<greater> ::= >
<greater_equal> ::= >=
<leads to> ::= =>
<equal> ::= ==
<unequal> ::= !=
<membership_in> ::= IN
<membership_not_in> ::= NOT IN
<underscore> ::= _
<asterisk> ::= *
<slash> ::= /
<any_character> ::= !! any character
<hash_symbol> ::= #
<dollar_sign> ::= $
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```

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

```

Identifiers

GraphScript uses identifiers to reference schema names, graph workspace names, and attribute names. Semantically, GraphScript identifiers are equivalent to SQL identifiers.

```

<simple_identifier> ::= ( <letter> | <underscore> ) { <letter> | <underscore> |
               <digit> | <hash_symbol> | <dollar_sign> }
<special_identifier> ::= <double_quote> { <any_charater> } <double_quote>
<identifier> ::= <simple_identifier> | <special_identifier>

```

Local Variable Expressions

GraphScript supports local variables that split large and complex statements into smaller ones, in order to improve the overall readability. After the initial assignment or declaration (see [Assignment Statements \[page 33\]](#) in [Statements \[page 32\]](#)), the type of a variable can no longer be modified. A local variable expression can be used at any place that allows an expression, as long as the corresponding expression types match. Variable names are case-insensitive. Variables that are used for reading access have a leading colon. Variables that are used for writing access have no leading colon.

```

<variable> ::= <letter> { <letter> | <digit> | <underscore> }
<variable_reference> ::= <colon> <letter> { <letter> | <digit> | <underscore> }
<local_var_expr> ::= <variable_reference>

```

Examples

```

aVariable
aSecondVariable12
:aVariable

```

Attribute Access Expressions

An attribute access expression allows you to retrieve the corresponding attribute value for a given local variable name and a given attribute or temporary attribute name. The attribute name is an identifier, which may be quoted. The return type of the expression is derived from the underlying specified attribute type.

```

<attr_access_expr> ::= <variable_reference> <dot> <identifier>

```

Examples

```

:i.weight
:i."weight"

```

```
:i."weight int"
```

Relational Expressions

Relational expressions are binary expressions that compare the values produced by two expressions against each other and return a Boolean value. Both expressions have to be of the same type. For types Vertex, Edge, and Boolean, only equality and inequality comparisons are allowed. For membership testing, the right operand must be a multiset.

```
<relational_op> ::= <equal> | <unequal> | <lower> | <greater> | <lower_equal> |  
  <greater_equal> | <membership_in> | <membership_not_in>  
<relational_expr> ::= <expr> <relational_op> <expr>
```

Examples

```
2 > 3  
3 == 4  
:a.weight > 3  
:a.weight NOT IN :set_of_values
```

Arithmetic Expressions

Arithmetic expressions are binary expressions that take two expressions and an arithmetic operator and produce a result. Both expressions have to be of the same type.

```
<arithmetic_op> ::= <plus> | <minus> | <asterisk> | <slash>  
<arithmetic_expr> ::= <expr> <arithmetic_op> <expr>
```

String Concatenation

String concatenation is a binary expression that takes two expressions and a concatenation operator and produces a combined string value. Both expressions have to be of the same character string type.

```
<concat_op_expr> ::= <expr> <pipe> <expr>
```

Examples

```
VARCHAR str1 = 'Basket';  
VARCHAR str2 = 'ball';  
VARCHAR str3 = :str1 || :str2;
```

Logical Expressions

Logical expressions are binary expressions that take two expressions and a logical operator and produce a Boolean result value. Both expressions have to be of the same type.

```
<logical_op> ::= AND | OR
<logical_expr> ::=
<expr> <logical_op> <expr>
| NOT <l_paren> <expr> <r_paren>
```

Examples

```
(3 == 4) AND (4 == 3)
TRUE OR (:a.weight > 3)
NOT (3 == 2)
```

Collection Initializer Expressions

A collection initializer expression can be used to initialize a multiset or a sequence. A multiset or a sequence of scalars can also be initialized from a table attribute access expression, provided that their element primitive types match.

```
<initializer_list> ::= <expr> | <initializer_list> <comma> <expr>
<multiset_init_expr> ::=
<l_curly> <initializer_list> <r_curly>
| <multiset_type> <l_paren> <attr_access_expr> <r_paren>
<sequence_init_expr> ::=
<l_square> <initializer_list> <r_square>
| <sequence_type> <l_paren> <attr_access_expr> <r_paren>
```

Examples

```
Graph g = Graph("MYWORKSPACE");
Multiset<Vertex> vm = { Vertex(:g,1), Vertex(:g,2), Vertex(:g,1) };
Sequence<Vertex> vs = [ Vertex(:g,1), Vertex(:g,2), Vertex(:g,1) ];
Multiset<Int> mi = Multiset<Int>( :tab."intCol" );
Sequence<Int> si = Sequence<Int>( :tab."intCol" );
```

Closure Expressions

A closure expression is an anonymous function with an optional environment, which allows you to capture variables defined in the same or an outer program scope. All variables from the same or an outer scope are implicitly captured by reference, that is, they can be modified within the closure expression. Furthermore, a closure expression takes a set of parameters, which are bound to actual values upon invocation. A closure expression defines a return type, which can be either specified implicitly (the default return type is Void) or explicitly. If the return type is different from Void, all reachable code paths must contain a valid return statement.

```
<return_type> ::= <type> | VOID
```

```

<clos_expr> ::= <l_paren> <typed_paramlist> <r_paren>
  [ <leads_to> <return_type> ] <l_curly> { <statement> } <r_curly>
<typed_paramlist> ::= <type> <variable> |
  <typed_paramlist> <comma> <type> <variable>

```

The body of the closure expression consists of a set of statements. Closure expressions cannot be bound to variables and can only be defined and used in place in the Shortest_Path built-in algorithm to specify a custom weight function or in traversal statements to define the actions upon invocation.

Examples

```

WeightedPath<Int> p = SHORTEST_PATH(:g, Vertex(:g, 1), Vertex(:g, 2),
  (Edge e) => INT { return :e.weight; });

```

Function Expressions

A function expression takes a list of parameters and produces a result of the specified type. A function can be overloaded on the parameters; in other words, there can be multiple function definitions with the same function name and different parameters. Overloading based solely on the function return type is not supported. For a complete list of supported built-in functions, see [Built-In Functions \[page 38\]](#).

```

<function_name> ::= <letter> { <letter> | <underscore> }
<function_expr> ::= <function_name> <l_paren> [ <parameter_list> ] <r_paren>
<parameter_list> ::= <expr> | <parameter_list> <comma> <expr>

```

Function expressions returning graph objects have a special role, because the graph objects logically contain their vertices and edges. Built-in functions returning graph objects can only be used in definition statements to initialize graph variables.

The functions Graph, Subgraph, Inversegraph and Shortest_Paths_One_To_All return graph objects. Shortest_Path returns a WeightedPath whereas Strongly_Connected_Components returns a Multiset<Multiset<Vertex>>.

Graph

Creates a graph object. Two different variations are available:

```

<graph_function> ::= 'GRAPH' <l_paren> <schema_name>, <workspace_name> <r_paren>
<schema_name> ::= <identifier>
<workspace_name> ::= <identifier>

```

Creates a graph object based on the specified workspace in the specified schema. If the schema parameter is omitted, then the SQL schema inference rules apply.

```

<graph_function> ::= 'GRAPH' <l_paren>
  <edge_table_variable>,
  <source_column_identifier>,
  <target_column_identifier>,
  <edge_id_column_identifier>,
  <vertex_table_variable>,
  <vertex_id_column_identifier> <r_paren>
<edge_table_variable> ::= <variable_reference>
<source_column_identifier> ::= <identifier>
<target_column_identifier> ::= <identifier>
<edge_id_column_identifier> ::= <identifier>

```



```
<vertex_table_variable> ::= <variable_reference>
<vertex_id_column_identifier> ::= <identifier>
```

Creates a graph object based on the specified edge and vertex tables. Besides the edge and vertex table variables, the column identifiers of edge source, edge target, edge key, and vertex key need to be specified.

Examples

```
Graph g1 = Graph("GREEK_MYTHOLOGY", "GRAPH");
Graph g2 = Graph(:edgeTable, "sourceCol", "targetCol", "edgeIdCol",
:vertexTable, "vertexIdCol");
```

Subgraph

```
<subgraph_function> ::= 'SUBGRAPH' <l_paren> <parent_graph_variable>,
<container_of_vertices> <r_paren>
<parent_graph_variable> ::= <variable_reference>
<container_of_vertices> ::= <expr>
```

Returns the subgraph of the given parent graph induced by the given container of vertices. The subgraph contains a copy of all vertices in the input vertex container as well as all edges of the parent graph whose source and target are both in the given vertex container.

```
<subgraph_function> ::= 'SUBGRAPH' <l_paren> <parent_graph_variable>,
<container_of_edges> <r_paren>
<parent_graph_variable> ::= <variable_reference>
<container_of_edges> ::= <expr>
```

Returns the subgraph of the given parent graph induced by the given container of edges. The subgraph contains a copy of the given edges as well as all source and target vertices of the given edge container.

All vertices and edges of a subgraph originate from the parent graph, but are different objects. A vertex or edge of a subgraph has all persistent and temporary attributes of a vertex or edge in the parent graph. Modifications of temporary attributes in the parent graph are not propagated to the corresponding temporary attributes of a subgraph and vice versa.

The parent graph of a subgraph can be a subgraph of another graph.

Examples

The following example demonstrates the usage of the subgraph function in combination with temporary attributes. The temporary attribute `pet` is defined on graph `G` before deriving the subgraph `OlympianGraph` from `G`. Thus, it is available in both graphs. As the temporary attribute `weapon` is added to `G` after creating `OlympianGraph`, accessing this vertex attribute for a vertex of `OlympianGraph` leads to an error, as demonstrated. We can, however, add a vertex attribute with the same name and type to `OlympianGraph`, rendering the previously erroneous statement valid. Although vertices `Zeus` and `Zeus2` share the same key, these are two separate objects and they have different values for the temporal attribute `pet`.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
ALTER G ADD TEMPORARY VERTEX ATTRIBUTE(VARCHAR pet = '');
Graph OlympianGraph = SUBGRAPH(
:G, V IN Vertices(:G) WHERE :V."RESIDENCE" == 'Olymp');
ALTER G ADD TEMPORARY VERTEX ATTRIBUTE(VARCHAR weapon = '');
Vertex Poseidon = Vertex(:OlympianGraph, 'Poseidon');
Poseidon.weapon = 'Trident'; -- ERROR:
attribute "weapon" is not defined for OlympianGraph
ALTER OlympianGraph ADD TEMPORARY VERTEX ATTRIBUTE(VARCHAR weapon = '');
Poseidon.weapon = 'Trident'; -- OK
Vertex Zeus = Vertex(:G, 'Zeus');
```

```
Vertex Zeus2 = Vertex(:OlympianGraph, 'Zeus');
Zeus.pet = 'Eagle';
Zeus2.pet = 'Thunderbird';
Graph MarriedOlympianGraph = SUBGRAPH(:OlympianGraph, E IN Edges(:OlympianGraph)
WHERE :E."TYPE" == 'marriedTo');
```

Inversegraph

```
<subgraph_function> ::=
  'INVERSEGRAPH' <l_paren> <parent_graph_variable> <r_paren>
<parent_graph_variable> ::= <variable_reference>
```

Returns the inverse graph of the given parent graph. All vertices of the parent graph are copied into the resulting inverse graph without any modification. All edges of the parent graph are copied into the resulting inverse graph with swapped edge direction. The values for all vertex as well as edge attributes are copied from the parent graph. Attribute handling is identical to subgraphs.

Although it is possible to derive an inverse graph from a subgraph, we do not support calling the SUBGRAPH and INVERSEGRAPH function on inverse graphs.

Examples

In the following example, we construct a multiset ZeusNeighbors of all direct neighbors of Zeus independently of the edge direction using INVERSEGRAPH. Note that we use a FOREACH loop to get the corresponding vertices in Graph G by key, as using the UNION operation for multisets from different graphs is not permitted.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
Multiset<Vertex> ZeusNeighbors = NEIGHBORS(:G, [ Vertex(:G, 'Zeus') ], 1, 1);
Graph GI = INVERSEGRAPH(:G);
FOREACH V IN NEIGHBORS(:GI, { Vertex(:GI, 'Zeus') }, 1, 1) {
  ZeusNeighbors = :ZeusNeighbors UNION { Vertex(:G, :V."NAME") };
}
```

The Neighbors function supports reverse traversal as well. A negative distance range specifies that the incoming edges rather than the outgoing edges are used for the traversal. The same result as above can be produced using a single Neighbors call with a corresponding distance range.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
Multiset<Vertex> ZeusNeighbors = NEIGHBORS(:G, [ Vertex(:G, 'Zeus') ], -1, 1);
```

Shortest Path

```
<sssp_function> ::= 'SHORTEST_PATH' <l_paren> <parent_graph_variable>,
<source_vertex>, <target_vertex> [ , <clos_expr> ] <r_paren>
<parent_graph_variable> ::= <variable_reference>
<start_vertex> ::= <expr>
<target_vertex> ::= <expr>
```

Returns a WeightedPath instance containing a shortest path within the given parent graph (first parameter) from a start vertex (second parameter) to a target vertex (third parameter). By default, the weight metric of the SHORTEST_PATH built-in function is hop distance, but it is also possible to supply a custom weight function (optional fourth parameter).

The weight function takes an edge as parameter and returns a value of a numerical type: INT, BIGINT, or DOUBLE. The return type matches the WeightedPath weight type. The weight function may contain only a single statement, which needs to be a return statement. Furthermore, it is only possible to return literals or to perform edge attribute access. It is currently not possible to access variables defined outside the weight function.

Like a subgraph, the `WeightedPath` contains copies of the corresponding vertices and edges from the parent graph. All vertices and edges of a `WeightedPath` originate from the parent graph, but are different objects. A vertex or edge of a `WeightedPath` has all persistent and temporary attributes of a vertex or edge in the parent graph at the time of path creation. Modifications of temporary attributes in the parent graph are not propagated to the corresponding temporary attributes of a `WeightedPath` and the other way round.

It is possible to define temporary attributes for the vertices and edges of a path.

Examples

The following example demonstrates the usage of the `SHORTEST_PATH` function. For a more in-depth explanation of the semantics for temporary attributes of vertices and edges in a path, see the subgraph example above.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
Vertex sourceVertex = Vertex(:G, 'Uranus');
Vertex targetVertex = Vertex(:G, 'Hephaestus');
WeightedPath<BigInt> p = SHORTEST_PATH(:G, :sourceVertex, :targetVertex);
Sequence<Vertex> vertices = VERTICES(:p);
Sequence<Edge> edges = EDGES(:p);
BigInt lengthOfP = Length(:p);
BigInt weightOfP = Weight(:p);
ALTER p ADD TEMPORARY VERTEX ATTRIBUTE (Int vertexAttr = 0);
ALTER p ADD TEMPORARY EDGE ATTRIBUTE (Int edgeAttr = 42);
-- Get first edge of path
Edge e = :edges[1L];
-- Read edge attribute value --> 42
Int edgeAttrValue = :e.edgeAttr;
```

The following example demonstrates the use of a custom weight function. The first path of type `WeightedPath<Int>` is constructed using a weight function depending on a temporary edge attribute. The two other paths use fixed weights of type `BigInt` and `Double`.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
Vertex sourceVertex = Vertex(:G, 'Uranus');
Vertex targetVertex = Vertex(:G, 'Hephaestus');
ALTER g ADD TEMPORARY EDGE ATTRIBUTE (Int weight = 1);
WeightedPath<Int> p = SHORTEST_PATH(
    :G, :sourceVertex, :targetVertex, (Edge e) => INT { return :e.weight;});
WeightedPath<BigInt> p2 = SHORTEST_PATH(
    :G, :sourceVertex, :targetVertex, (Edge e) => BIGINT{ return 42L;});
WeightedPath<Double> p3 = SHORTEST_PATH(
    :G, :sourceVertex, :targetVertex, (Edge e) => DOUBLE{ return 1.1;});
```

Shortest paths one to all

```
<spoa_function> ::= SHORTEST_PATHS_ONE_TO_ALL <l_paren> <parent_graph_variable>,
    <source_vertex>, <distance_attribute_name> [ , <clos_expr> ] <r_paren>
<parent_graph_variable> ::= <variable_reference>
<start_vertex> ::= <expr>
<distance_attribute_name> ::= <identifier>
```

Returns a sub-graph of the given parent graph (first parameter) built from the shortest paths from the start vertex (second parameter) to all other reachable vertices. It contains all the attributes from the parent graph and an additional vertex attribute (third parameter) containing the distance from the start vertex to each reachable vertex. The type of the distance attribute is either `BIGINT` or the return type of the optional weight function. By default, the weight metric of the `SHORTEST_PATHS_ONE_TO_ALL` built-in function is hop distance, but it is also possible to supply a custom weight function (optional fourth parameter). See [Shortest Path](#) for details on the weight function. The resulting sub-graph is a tree, i.e. a directed, acyclic graph where each vertex has only one incoming edge, except for the start vertex, which has no incoming edge.

Examples

The following example demonstrates the usage of the SHORTEST_PATHS_ONE_TO_ALL function.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
Vertex sourceVertex = Vertex(:G, 'Uranus');
Graph hierarchy = SHORTEST_PATHS_ONE_TO_ALL(:G, :sourceVertex, "distance");
```

The following example demonstrates the use of a custom weight function. The first graph is constructed using a weight function depending on a temporary edge attribute. The two other graphs use fixed weights of type BigInt and Double.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
Vertex sourceVertex = Vertex(:G, 'Uranus');
ALTER g ADD TEMPORARY EDGE ATTRIBUTE (Int weight = 1);
Graph hierarchy = SHORTEST_PATHS_ONE_TO_ALL(
    :G, :sourceVertex, "distance", (Edge e) => INT { return :e.weight;});
Graph hierarchy = SHORTEST_PATHS_ONE_TO_ALL(
    :G, :sourceVertex, "distance", (Edge e) => BIGINT{ return 42L;});
Graph hierarchy = SHORTEST_PATHS_ONE_TO_ALL(
    :G, :sourceVertex, "distance", (Edge e) => DOUBLE{ return 1.1;});
```

K-shortest paths

```
<ksp_function> ::= 'K_SHORTEST_PATHS' <l_paren> <parent_graph_variable>,
<source_vertex>, <target_vertex>, <num_paths> [ , <clos_expr> ] <r_paren>
<parent_graph_variable> ::= <variable_reference>
<start_vertex> ::= <expr>
<target_vertex> ::= <expr>
<num_paths> ::= <expr>
```

Returns a sequence of WeightedPath instances containing the num_paths (fourth parameter) shortest paths within the given parent graph from a start vertex (second parameter) to a target vertex (third parameter). By default, the weight metric of the K_SHORTEST_PATHS built-in function is hop distance, but it is also possible to supply a custom weight function (optional fourth parameter). See [Shortest Path](#) for details on the weight function. The function creates paths that might contain cycles. The paths in the sequence are ordered by increasing length.

Examples

The following example demonstrates the usage of the K_SHORTEST_PATHS function.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
Vertex sourceVertex = Vertex(:G, 'Uranus');
Vertex targetVertex = Vertex(:G, 'Hephaestus');
Sequence<WeightedPath<BigInt>> paths = K_SHORTEST_PATHS(
    :G, :sourceVertex, :targetVertex, 3);
FOREACH p IN :paths {
    -- process the path
    BigInt l = LENGTH(:p);
    BigInt w = WEIGHT(:p);
    FOREACH v in VERTICES(:p) {
        -- ....
    }
}
```

The following example demonstrates the use of a custom weight function. The first path of type WeightedPath<Int> is constructed using a weight function depending on a temporary edge attribute. The two other paths use fixed weights of type BigInt and Double.

```
Graph G = Graph("GREEK_MYTHOLOGY", "GRAPH");
```

```

Vertex sourceVertex = Vertex(:G, 'Uranus');
Vertex targetVertex = Vertex(:G, 'Hephaestus');
ALTER g ADD TEMPORARY EDGE ATTRIBUTE (Int weight = 1);
Sequence<WeightedPath<Int>> paths = K_SHORTEST_PATHS(
    :G, :sourceVertex, :targetVertex, 3, (Edge e) => INT { return :e.weight;});
Sequence<WeightedPath<BigInt>> paths = K_SHORTEST_PATHS(
    :G, :sourceVertex, :targetVertex, 3, (Edge e) => BIGINT{ return 42L;});
Sequence<WeightedPath<Double>> paths = K_SHORTEST_PATHS(
    :G, :sourceVertex, :targetVertex, 3, (Edge e) => DOUBLE{ return 1.1;});

```

Strongly connected components

```

<scc_function> ::= 'STRONGLY_CONNECTED_COMPONENTS' <l_paren>
    <parent_graph_variable> <r_paren>
<parent_graph_variable> ::= <variable_reference>

```

Returns a multiset of multisets of vertices, representing the strongly connected components in the parent graph. The foreach loop statement can be used to iterate over the result and access the individual components. Each strongly connected component is represented as a multiset of vertices and can be used in the same way as any other multiset of vertices.

Examples

The following example demonstrates the usage of the strongly connected components function. The result of the function is a container of containers. The built-in function COUNT can be used to determine the total number of strongly connected components in the result. One can iterate over the result and use a temporary vertex attribute to store the component number to which the vertex belongs.

```

Graph g = Graph("GREEK_MYTHOLOGY", "GRAPH");
ALTER g ADD TEMPORARY VERTEX ATTRIBUTE(Int componentNumber);
Multiset<Multiset<Vertex>> sccResult = STRONGLY_CONNECTED_COMPONENTS(:g);
BigInt numComponents = COUNT(:sccResult);
Int componentCounter = 0;
FOREACH component IN :sccResult {
    componentCounter = :componentCounter + 1;
    FOREACH v IN :component {
        v.componentNumber = :componentCounter;
    }
}

```

Filter Expressions

A filter expression evaluates a predicate for each element of an input container. The matching elements are returned as a multiset. Within the WHERE clause of the filter expression, it is possible to access temporary variables, and temporary and non-temporary attributes of vertices and edges, to use built-in functions, and to call functions. For filter expressions on sequence containers, an optional ordinal variable can be used to access the ordinal position of the element currently being processed.

```

<filter_expr> ::= <variable> IN <expr> [ WITH ORDINALITY AS <variable> ]
    WHERE <expr>

```

Examples

```

INT tmp = 2;
v IN Vertices(:g) WHERE :v.color == 'green' AND :v.numEyes != 2;

```

```
v IN Vertices(:g) WHERE :v.id <= COUNT(Vertices(:g)) AND :v.numEyes == :tmp;
v IN [ Vertex(:g, 1), Vertex(:g, 2) ] WITH ORDINALITY AS i WHERE :i > 2;
```

Is Null Expressions

An Is Null expression allows you to check whether a value has the value null or not. The return type of the expression is a Boolean value.

```
<is_null_expression> ::= <expr> IS NULL
<is_not_null_expression> ::= <expr> IS NOT NULL
```

Examples

```
BOOLEAN is_null = :i."weight" IS NULL;
```

Set Operations

Set operations are binary expressions that take two expressions and a set operator and produce a multiset. Both expressions have to produce or be a multiset from the same graph.

```
<set_op> ::= <union> | <union_all> | <intersect> | <intersect_all> |
<except> | <except_all>
<set_op_expr> ::= <expr> <set_op> <expr>
```

```
Multiset<Vertex> vertices3 = :vertices1 UNION :vertices2;
vertices3 = :vertices3 UNION ALL {:v1};
```

Union all

Returns a multiset containing all entries from both expressions. Duplicates are not removed.

Union

Returns all unique entries from both expressions by removing the duplicates found. UNION has the same function as UNION ALL with a subsequent DISTINCT.

```
MULTISET<VERTEX> ms2 = {:v, :v2};
MULTISET<VERTEX> ms1 = {:v};
MULTISET<VERTEX> ms3 = :ms1 UNION :ms2 UNION {:v3};
```

Intersect all

Returns all entries that are present in both expressions. Duplicates are not removed.

Intersect

Returns all unique entries that are present in both expressions. Duplicates are removed. Intersect has the same function as INTERSECT ALL with a subsequent DISTINCT.

Except all

Returns all entries that are present in the left-hand expression but not in the right-hand expression. Duplicates are not removed.

```
MULTISET<VERTEX> ms2 = { :v, :v2 };
MULTISET<VERTEX> ms1 = { :v };
MULTISET<VERTEX> ms3 = :ms1 EXCEPT ALL :ms2;
```

Except

Returns all unique entries that are present in the left-hand expression but not in the right-hand expression. Duplicates are removed. Except has the same function as Except ALL with a subsequent DISTINCT.

```
MULTISET<VERTEX> ms2 = { :v, :v2 };
MULTISET<VERTEX> ms1 = { :v };
MULTISET<VERTEX> ms3 = :ms1 EXCEPT :ms2;
```

Sequence Operations

The sequence type can be used in positional access (`index_expr`) and concatenation (`concat_expr`) operations.

```
<concat_expr> ::= <expr> || <expr>
<index_expr> ::= <expr><l_square><expr><r_square>
```

Read index access

Reads the value at a certain position of a sequence. Throws an `OutOfBoundsException` if the index is larger than the capacity of the sequence. Sequence indices start at 1L.

```
Sequence<Vertex> vs = [ :v, :v2 ];
Vertex v3 = :vs[2L];
```

Write index access

Replaces a value at a certain index within a sequence. Throws an `OutOfBoundsException` if the index is larger than the capacity of the sequence.

```
Sequence<Vertex> vs = [ :v, :v2 ];
vs[3L] = :v;
```

Concatenation

Concatenates two sequences to form a new sequence. In case of vertices and edges, concatenation of a single element is possible.

```
<concat_op_expr> ::= <expr> <pipe> <expr>
```

```
Sequence<Vertex> vs = [ :v, :v2 ];
Sequence<Vertex> vs2 = :vs || [ :v ];
Sequence<Vertex> vs3 = :vs || :v;
```

Table Operations

The table type can be used in cell access (`cell_expr`) operations. The row count of a table can be determined using the COUNT function expression.

Cell read access

Reads the value at a certain index position of a specified table column. Returns NULL if the index is larger than the row count of the table. Cell indices start at 1L.

```
Int i = :tab."IntColumn"[1L];
```

Cell write access

Sets the value at a certain index position of a specified table column. If the index is larger than the row count of the table, then the table is resized accordingly and padded with NULL values if necessary.

```
tab."IntColumn"[5L] = 123;
```

Row count

Reads the number of rows of a specified table.

```
BigInt numRows = COUNT(:tab);
```

Related Information

[Built-In Functions \[page 38\]](#)

[Statements \[page 32\]](#)

6.7 Statements

The following types of statements are supported in GraphScript:

```
<statement> ::=  
<definition>  
| <assignment>  
| <conditional>  
| <foreach_loop>  
| <temp_attribute>  
| <while_loop>  
| <traversal>  
| <end_traversal>
```


Temporary Attribute Definition Statements

A temporary attribute can be used to store vertex-centric or edge-centric state information at script runtime. Read access to temporary attributes is the same as for persistent attributes; write access is only allowed for temporary attributes.

```
<attr_spec> ::= <primitive_type> <identifier> [ <equals> <expr> ]  
<attr_list> ::= <attr_spec> | <attr_list> <comma> <attr_spec>  
<temp_attribute> ::= "ALTER" <identifier> "ADD" "TEMPORARY" <vertex_or_edge>  
"ATTRIBUTE" <l_paren> <attr_list> <r_paren> <semicolon>
```

Definition Statements

A definition statement declares and defines a local variable of a specific type. Objects of primitive types are default-initialized to zero (for numeric values) or to an empty string (for character-based objects). BOOLEAN values are default-initialized to FALSE. Vertices, edges, and graphs have to be initialized in the declaration. A variable is visible and accessible in the current scope and all inner scopes, such as in loops and conditional statements. Variables defined inside loops and conditional statements are not accessible outside the statements. GraphScript does not support variable shadowing; in other words, a variable defined in an outer scope cannot be redefined in an inner scope.

```
<definition> ::= <vertex_or_edge> <variable> <equals> <expression> <semicolon>  
| <primitive_type> <variable> [ <equals> <expression> ] <semicolon>  
| <graph> <variable> <equals> <expression> <semicolon>  
| <multiset_type> <variable> <equals> <expression> <semicolon>  
| <sequence_type> <variable> <equals> <expression> <semicolon>
```

Examples

```
Graph g1 = Graph("MYSCHEMA", "MYWORKSPACE");  
Graph g2 = Graph("MYWORKSPACE");  
Graph g3 = Graph(MYWORKSPACE);  
Vertex v = Vertex(:g1, 1);  
Edge e = Edge(:g1, '3');  
Int i;  
Double d = 23.5;  
Varchar s = 'Dave';
```

Assignment Statements

An assignment statement binds the result of an expression to a local variable. Alternatively, if the left-hand side of the assignment statement is an attribute access expression, the value of the right-hand side expression is assigned to the corresponding attribute.

```
<assignment> ::= <variable> <equals> <expr> <semicolon>  
| <variable> <equals> <projection_expr> <semicolon>  
| <attr_access_expr> <equals> <expr> <semicolon>
```

GraphScript is statically typed and the data type of the object that is bound to the variable cannot change during the lifetime of a single query run. All variables have to be initialized before they can be referenced and accessed in a GraphScript expression.

Examples

```
Graph g = Graph("MYWORKSPACE");
Int i;
i = 23;
Vertex v = Vertex(:g,1);
v.attr = 23;
```

Projection Expression

Projection expressions are special expressions that can only be used in assignment statements. The variable of the assignment statement must be of type table and an output parameter. Projection expressions expect a container of vertices or a container of edges as an input in addition to an enumeration of projected attributes. For projection expressions on sequence containers, an optional ordinal variable can be used to access the ordinal position of the element currently being processed.

```
<attr_list> ::= <variable_reference> <dot> <identifier>
<projection_expr> ::= SELECT <attr_list> FOREACH <variable> IN <expr>
    [WITH ORDINALITY AS <variable>]
```

The order of the projected attributes and their types must match the columns of the output parameter type. Attribute names are ignored and the columns of the output table will be named according to the output parameter type.

```
myTable = SELECT :v."NAME", :v.visited FOREACH v in :myVertices;
myTable = SELECT :v."NAME", :i FOREACH v in [ Vertex(:g, 1)]
    WITH ORDINALITY AS i;
```

Note

- Tables or table types are permitted for IN and OUT parameters.
- Table types need to be created before they are used as parameter types. Inline definitions of table types in the parameter list are not permitted.

Examples

```
CREATE TYPE tt AS TABLE(NAME VARCHAR(1000), ATTR INT);
CREATE PROCEDURE myTableProc (IN inTab tt, OUT outTab tt)
LANGUAGE GRAPH READS SQL DATA AS
BEGIN
    Int i = :inTab.attr[1L];
    GRAPH g = GRAPH("GREEK_MYTHOLOGY", "GRAPH");
    ALTER g ADD TEMPORARY VERTEX ATTRIBUTE(INT ATTR = :i);
    outTab = SELECT :v."NAME", :v.attr FOREACH v in VERTICES(:g);
END;
```

Conditional Statements

A conditional statement consists of an IF body and an optional ELSE body. Each body can contain a list of statements (which may be empty), and the list of statements is enclosed in curly braces. If the expression <expr> in the IF branch evaluates to TRUE, the corresponding statements in the IF body are executed, otherwise the statements in the ELSE branch are executed.

```
<conditional> ::= IF <l_paren> <expr> <r_paren>
  <l_curly> { <statement> } <r_curly>
  [ ELSE <l_curly> { <statement> } <r_curly> ]
```

Examples

```
IF (2 < 3) {
  IF (TRUE) {
    INT I;
  } ELSE {
    INT I;
  }
}
```

Foreach Loop Statements

A foreach loop iterates over a container of vertices or edges. The iteration order is not specified and should not be assumed to be the same for two different script executions. In the loop body, a list of statements can be specified, which can reference the defined variable in the loop header. The variable scope of the variable is defined as the inner body of the loop statement; in other words, the variable can no longer be referenced after the loop statement. Within a foreach loop, the keywords break and continue can be used. For foreach loop statements on sequence containers, an optional ordinal variable can be used to access the ordinal position of the element currently being processed. Additionally, a foreach loop statement can be used to iterate over the result of the STRONGLY_CONNECTED_COMPONENT function, which is a multiset of multiset of vertices.

```
<foreach_loop> ::= FOREACH
  [ <l_paren> ] <variable> IN <expr> [WITH ORDINALITY AS <variable>]
  [ <r_paren> ] <l_curly> { <statement> } <r_curly>
```

Examples

```
Graph g = Graph("MYWORKSPACE");
FOREACH v IN { Vertex(:g,1), Vertex(:g,2) } { INT a = :v.attr;
}
FOREACH v IN [ Vertex(:g,1), Vertex(:g,2) ] WITH ORDINALITY AS i {
  INT a = :i;
}
```

While Loop Statements

A while loop repeats the statements specified in the loop body as long as the condition in the loop head evaluates to true. Within a while loop, the keywords `break` and `continue` can be used. `break` stops the further execution of the loop and `continue` skips to the next loop iteration.

```
<while_loop> ::= WHILE <l_paren> <expr> <r_paren>  
<l_curly> { <statement> } <r_curly>
```

Examples

```
INT a = 10;  
WHILE (:a > 0) {  
    a = :a - 1;  
}
```

Traversal Statements

The traversal statement explores the graph in a breadth-first manner and allows specifying operations to be performed at each visit of a vertex or an edge during the traversal. The set of operations to be performed is called a traversal hook. A traversal statement can have any non-empty combination of vertex and edge traversal hooks, where vertex traversal hooks and edge traversal hooks may only appear once respectively. Accepted hook signatures can be one of the following parameter combinations: (Vertex) => Void, (Edge) => Void, (Vertex, BigInt) => Void, or (Edge, BigInt) => Void. The optional second parameter of type `BigInt` corresponds to the traversal depth. The traversal depth is zero-based, that is, the start vertex or vertices have traversal depth 0.

Variables defined outside a traversal hook can be accessed and modified within the hook.

```
<traversal> ::= TRAVERSE BFS <expr> FROM <expr> <hook> { <hook> } <semicolon>  
<hook> ::= ON VISIT VERTEX <clos_expr>  
         | ON VISIT EDGE <clos_expr>
```

Examples

```
TRAVERSE BFS :g FROM { Vertex(:g, 1), Vertex(:g, 2) }  
    ON VISIT VERTEX (Vertex v) {  
        Int I = :v.attr;  
    };  
TRAVERSE BFS :g FROM Vertex(:g, 1)  
    ON VISIT EDGE (Edge e) {  
        Int I = :e.attr;  
    };  
TRAVERSE BFS :g FROM Vertex(:g, 1)  
    ON VISIT VERTEX (Vertex v) {  
        Int I = :v.attr;  
    }  
    ON VISIT EDGE (Edge e) {  
        Int I = :e.attr;  
    };  
TRAVERSE BFS :g FROM Vertex(:g, 1)  
    ON VISIT VERTEX (Vertex v, BigInt lvl) {  
        v.attr = :lvl;  
    }
```

```

ON VISIT EDGE (Edge e, BigInt lvl) {
    e.attr = :lvl;
};

```

END TRAVERSE ALL Statement

The END TRAVERSE ALL statement is only valid inside a traversal hook. When executed, the traversal is terminated. In case of BFS traversal, this means that no further levels of vertices or edges of the graph are explored. However, all hooks corresponding to vertices or edges on the current level are still completely executed.

```

<end_traversal> ::= END TRAVERSE ALL <semicolon>

```

Examples

```

TRAVERSE BFS :g FROM { Vertex(:g, 1), Vertex(:g, 2) }
ON VISIT VERTEX (Vertex v) {
    Int I = :v.attr;
    IF (:I > 5) {
        END TRAVERSE ALL;
    }
}
ON VISIT EDGE (Edge e) {
    Int I = :e.attr;
    IF (:I > 5) {
        END TRAVERSE ALL;
    }
};

```

END TRAVERSE Statement

The END TRAVERSE statement is only valid inside a traversal hook. When executed, the traversal is stopped at the current vertex or edge. In case of BFS traversal, this means the following:

- If END TRAVERSE is called from a vertex hook, then all outgoing edges of that vertex are ignored.
- If END TRAVERSE is called from an edge hook, then the target vertex of that edge will not be reached via this edge. However, it is still possible to reach the target vertex by a different edge.

```

<end_traversal> ::= END TRAVERSE ALL <semicolon>

```

Examples

```

TRAVERSE BFS :g FROM { Vertex(:g, 1), Vertex(:g, 2) }
ON VISIT VERTEX (Vertex v) {
    Int I = :v.attr;
    IF (:I > 5) {
        END TRAVERSE;
    }
}
ON VISIT EDGE (Edge e) {
    Int I = :e.attr;
    IF (:I > 5) {
        END TRAVERSE;
    }
};

```

6.8 Built-In Functions

The following table provides a summary of the built-in functions available in GraphScript.

Function Name	Description	Function Signature	Return Type
VERTEX	Constructs a vertex from a graph and vertex key.	Vertex(Graph, Int) Vertex(Graph, BigInt) Vertex(Graph, Varchar) Vertex(Graph, Nvarchar)	Vertex
EDGE	Constructs an edge from a graph and edge key.	Edge(Graph, Int) Edge(Graph, BigInt) Edge(Graph, Varchar) Edge(Graph, Nvarchar)	Edge
GRAPH	Constructs a graph from a graph workspace.	Graph(Identifier) Graph(Identifier, Identifier)	Graph
INVERSEGRAPH	Constructs an inverse graph from a graph.	InverseGraph(Graph)	Graph
SUBGRAPH	Constructs a subgraph of a graph induced by a set of vertices or edges.	Subgraph(Graph, Multiset) Subgraph(Graph, Sequence)	Graph
SHORTEST_PATH	Calculates a shortest path from start to target vertex.	Shortest_Path (Graph, Vertex, Vertex)	WeightedPath<BigInt>
SHORT-EST_PATHS_ONE_TO_ALL	Calculates the shortest paths from start to all other reachable vertices.	Shortest_Paths_One_To_All (Graph, Vertex, Varchar)	Graph
K_SHORTEST_PATHS	Calculates k shortest paths from start to target vertex.	K_Shortest_Paths (Graph, Vertex, Vertex, Int)	Sequence<Weighted-Path<BigInt>>
WEIGHT	Calculates the weight for a WeightedPath.	Weight(WeightedPath<BigInt>) Weight(WeightedPath<Int>) Weight(WeightedPath<Double>)	BigInt

Function Name	Description	Function Signature	Return Type
LENGTH	Calculates the length (number of edges) for a Weighted-Path.	Length(WeightedPath<Bi- gInt>) Length(WeightedPath<Int>) Length(WeightedPath<Dou- ble>)	BigInt
MULTISET	Constructs an empty multi- set of vertices or edges for a graph. For other types an empty multiset is not possi- ble.	Multiset(Graph)	Multiset
SEQUENCE	Constructs an empty se- quence of vertices or edges for a graph. For other types an empty sequence is not possible.	Sequence(Graph)	Sequence
COUNT	Counts the elements in a container.	Count(Multiset) Count(Sequence) Count(Multiset<Multi- set<Vertex>>)	BigInt
DISTINCT	Removes duplicates from a multiset.	Distinct(Multiset)	Multiset
SOURCE	Returns the source vertex of an edge.	Source(Edge)	Vertex
TARGET	Returns the target vertex of an edge.	Target(Edge)	Vertex
IN_DEGREE	Returns the number of in- coming edges of a vertex.	In_Degree(Vertex)	BigInt
OUT_DEGREE	Returns the number of out- going edges of a vertex.	Out_Degree(Vertex)	BigInt
DEGREE	Returns the number of in- coming and outgoing edges of a vertex.	Degree(Vertex)	BigInt
IN_EDGES	Returns all the incoming edges of a vertex.	In_Edges(Vertex)	Multiset
OUT_EDGES	Returns all the outgoing edges of a vertex.	Out_Edges(Vertex)	Multiset
EDGES	Returns all the incoming and outgoing edges of a vertex.	Edges(Vertex)	Multiset
IS_REACHABLE	Returns whether there is a path from vertex v1 to vertex v2.	Is_Reachable(Graph, Vertex, Vertex)	Bool

Function Name	Description	Function Signature	Return Type
NEIGHBORS	Returns all reachable vertices from a start vertex or set of start vertices in the given distance range. A positive distance range specifies that the outgoing edges are used for the traversal. A negative distance range specifies that the incoming edges are used for the traversal. Distance ranges with a negative left bound and a positive right bound are supported as well.	Neighbors(Graph, Vertex, Int, Int) Neighbors(Graph, Multiset, Int, Int) Neighbors(Graph, Sequence, Int, Int)	Multiset
STRONGLY_CONNECTED_COMPONENTS	Computes the strongly connected components in a graph.	Strongly_Connected_Components(Graph)	Multiset<Multiset<Vertex>>
VERTICES	Returns all vertices in a graph or path.	Vertices(Graph) Vertices(WeightedPath)	Sequence
EDGES	Returns all edges in a graph or path.	Edges(Graph) Edges(WeightedPath)	Sequence
EDGES	Returns all edges between a start set or vertex and a target set or vertex in a graph.	Edges(Graph, Multiset, Multiset) Edges(Graph, Multiset, Vertex) Edges(Graph, Vertex, Multiset) Edges(Graph, Vertex, Vertex) Edges(Graph, Sequence, Sequence) Edges(Graph, Sequence, Multiset) Edges(Graph, Multiset, Sequence) Edges(Graph, Sequence, Vertex) Edges(Graph, Vertex, Sequence)	Multiset
INT	Casts a double or bigint value to an integer value. Throws a conversion error in the event of value overflows or underflows.	Int(Int) Int(Double) Int(BigInt)	Int

Function Name	Description	Function Signature	Return Type
INTEGER	Casts a double or bigint value to an integer value. Throws a conversion error in the event of value overflows or underflows.	Integer(Int) Integer(Double) Integer(BigInt)	Integer
DOUBLE	Casts an integer or bigint value to a double value.	Double(Double) Double(Int) Double(BigInt)	Double
BIGINT	Casts an integer or double value to a bigint value. Throws a conversion error in the event of value overflows or underflows.	BigInt(BigInt) BigInt(Int) BigInt(Double)	BigInt
VARCHAR	Casts an Nvarchar value to a Vchar value. The second parameter specifies the length of the Vchar type that is returned. Throws a conversion error if the Nvarchar value is too large to fit into the return type.	Vchar(Nvarchar, Int)	Vchar
NVARCHAR	Casts a Vchar value to an Nvarchar value. The second parameter specifies the length of the Nvarchar type that is returned. Throws a conversion error if the Vchar value is too large to fit into the return type.	Nvarchar(Vchar, Int)	Nvarchar
MULTISET	Casts a Sequence to a Multiset. The order of the elements is lost.	Multiset(Sequence)	Multiset
SEQUENCE	Casts a Multiset to a Sequence.	Sequence(Multiset)	Sequence
TIMESTAMP	Creates a timestamp value from the given character string. Throws a conversion error if the character string is not a valid timestamp value.	Timestamp(Vchar) Timestamp(NVchar)	Timestamp
NANOSECOND	Extracts the nanosecond component from a timestamp value.	Nanosecond(Timestamp)	Int
SECOND	Extracts the second component from a timestamp value.	Second(Timestamp)	Int
MINUTE	Extracts the minute component from a timestamp value.	Minute(Timestamp)	Int

Function Name	Description	Function Signature	Return Type
HOUR	Extracts the hour component from a timestamp value.	Hour(Timestamp)	Int
DAYOFMONTH	Extracts the day-of-month component from a timestamp value.	DayOfMonth(Timestamp)	Int
MONTH	Extracts the month component from a timestamp value.	Month(Timestamp)	Int
YEAR	Extracts the year component from a timestamp value.	Year(Timestamp)	Int

6.9 Reserved Keywords

Reserved words are words, which have a special meaning to the GraphScript parser in the SAP HANA database and cannot be used as variable names and identifiers.

Reserved words should not be used in GraphScript statements for schema and graph workspace object names. If necessary, you can work around this limitation by delimiting a schema name, a column name, or a graph workspace name in double quotation marks.

The following table lists all the current reserved words for the GraphScript:

ABS	ABSOLUTE	ACTION
ADA	ADD	ADMIN
AFTER	ALL	ALLOCATE
ALTER	ALWAYS	AND
ANY	ARE	ARRAY_AGG
ARRAY_MAX_CARDINALITY	ARRAY	AS
ASC	ASENSITIVE	ASSERTION
ASSIGNMENT	ASYMMETRIC	AT
ATOMIC	ATTRIBUTE	ATTRIBUTES
AUTHORIZATION	AVG	
BEFORE	BEGIN_FRAME	BEGIN_PARTITION
BEGIN	BERNOULLI	BETWEEN
BIGINT	BINARY	BLOB
BOOL	BOOLEAN	BOTH
BREADTH	BREAK	BY
CALL	CALLED	CARDINALITY
CASCADE	CASCADED	CASE

CAST	CATALOG_NAME	CEIL
CEILING	CHAIN	CHAR_LENGTH
CHAR	CHARACTER_LENGTH	CHARACTER_SET_CATALOG
CHARACTER_SET_NAME	CHARACTER_SET_SCHEMA	CHARACTER
CHARACTERISTICS	CHARACTERS	CHECK
CLASS_ORIGIN	CLOB	CLOSE
COALESCE	COBOL	COLLATE
COLLATION_CATALOG	COLLATION_NAME	COLLATION_SCHEMA
COLLATION	COLLECT	COLUMN_NAME
COLUMN	COMMAND_FUNCTION_CODE	COMMAND_FUNCTION
COMMIT	COMMITTED	CONDITION_NUMBER
CONDITION	CONNECT	CONNECTION_NAME
CONNECTION	CONST	CONSTRAINT_CATALOG
CONSTRAINT_NAME	CONSTRAINT_SCHEMA	CONSTRAINT
CONSTRAINTS	CONSTRUCTOR	CONTAINS
CONTINUE	CONTINUE	CONVERT
CORR	CORRESPONDING	COUNT
COVAR_POP	COVAR_SAMP	CREATE
CROSS	CUBE	CUME_DIST
CURRENT_CATALOG	CURRENT_DATE	CURRENT_DEFAULT_TRANSFORM_GROUP
CURRENT_PATHCURRENT_ROLE	CURRENT_TRANSFORM_GROUP_FOR_TYPE	CURRENT_USER
CURRENT	CURSOR_NAME	CURSOR
CYCLE		
DATE	DAY	DEALLOCATE
DEC	DOUBLE	DECIMAL
DECLARE	DEFAULT	DELETE
DO	DENSE_RANK	DEREF
DESCRIBE	DETERMINISTIC	DISCONNECT
DISTINCT	DROP	DYNAMIC
DATA	DATETIME_INTERVAL_CODE	DATETIME_INTERVAL_PRECISION
DEFAULTS	DEFERRABLE	DEFERRED
DEFINED	DEFINER	DEGREE
DEPTH	DERIVED	DESC
DESCRIPTOR	DIAGNOSTICS	DISPATCH

DOMAIN	DYNAMIC_FUNCTION	DYNAMIC_FUNCTION_CODE
EACH	EDGE	ELEMENT
ELSE	END_FRAME	END_PARTITION
END	END-EXEC	ENFORCED
ENUM	EQUALS	ESCAPE
EVERY	EXCEPT	EXCLUDE
EXCLUDING	EXEC	EXECUTE
EXISTS	EXP	EXPRESSION
EXTERNAL	EXTRACT	
FALSE	FETCH	FILTER
FINAL	FIRST_VALUE	FIRST
FLAG	FLOAT	FLOOR
FOLLOWING	FOR	FOREACH
FOREIGN	FORTRAN	FOUND
FRAME_ROW	FREE	FROM
FULL	FUNCTION	FUSION
GENERAL	GENERATED	GET
GLOBAL	GO	GOTO
GRANT	GRANTED	GRAPH
GROUP	GROUPING	GROUPS
HAVING	HIERARCHY	HOLD
HOOK	HOUR	
IDENTITY	IF	IGNORE
IMMEDIATE	IMMEDIATELY	IMPLEMENTATION
IMPORT	IN	INCLUDE
INCLUDING	INCREMENT	INDICATOR
INITIALLY	INNER	INOUT
INPUT	INSENSITIVE	INSERT
INSTANCE	INSTANTIABLE	INSTEAD
INT	INTEGER	INTERSECT
INTERSECTION	INTERVAL	INTO
INVOKER	IS	ISOLATION
JOIN		
KEY	KEY_MEMBER	KEY_TYPE
LAG	LANGUAGE	LARGE
LAST_VALUE	LAST	LATERAL

LEAD	LEADING	LEFT
LENGTH	LEVEL	LIKE_REGEX
LIKE	LIST	LN
LOCAL	LOCALTIME	LOCALTIMESTAMP
LOCATOR	LOWER	
MAP	MATCH	MATCHED
MAX	MAXVALUE	MEMBER
MERGE	MESSAGE_LENGTH	MESSAGE_OCTET_LENGTH
MESSAGE_TEXT	METHOD	MIN
MINUTE	MINVALUE	MOD
MODIFIES	MODULE	MONTH
MORE	MULTISET	MUMPS
NAME	NAMES	NAMESPACE
NATIONAL	NATURAL	NCHAR
NCLOB	NESTING	NEW
NEXT	NFC	NFD
NFKC	NFKD	NO
NONE	NORMALIZE	NORMALIZED
NOT	NTH_VALUE	NTILE
NULL	NULLABLE	NULLIF
NULLS	NUMBER	NUMERIC
NVARCHAR		
OBJECT	OCCURRENCES_REGEX	OCTET_LENGTH
OCTETS	OF	OFFSET
OLD	ON	ONLY
OPEN	OPTION	OPTIONS
OR	ORDER	ORDERING
ORDINALITY	OTHERS	OUT
OUTER	OUTPUT	OVER
OVERLAPS	OVERLAY	OVERRIDING
PAD	PARAMETER_MODE	PARAMETER_NAME
PARAMETER_ORDINAL_POSITION	PARAMETER_SPECIFIC_CATALOG	PARAMETER_SPECIFIC_NAME
PARAMETER_SPECIFIC_SCHEMA	PARAMETER	PARTIAL
PARTITION	PASCAL	PATH
PERCENT_RANK	PERCENT	PERCENTILE_CONT
PERCENTILE_DISC	PERIOD	PERSISTENT

PLACING	PLI	PORTION
POSITION_REGEX	POSITION	POWER
PRECEDES	PRECEDING	PRECISION
PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVILEGES	PROCEDURE
PUBLIC		
RANGE	RANK	READ
READS	REAL	RECURSIVE
REF	REFERENCES	REFERENCING
REGR_AVGX	REGR_AVGY	REGR_COUNT
REGR_INTERCEPT	REGR_R2	REGR_SLOPE
REGR_SXX	REGR_SXY	REGR_SYY
RELATIVE	RELEASE	REPEATABLE
RESPECT	RESTART	RESTRICT
RESULT	RETURN	RETURNED_CARDINALITY
RETURNED_LENGTH	RETURNED_OCTET_LENGTH	RETURNED_SQLSTATE
RETURNS	REVOKE	RIGHT
ROLE	ROLLBACK	ROLLUP
ROUTINE_CATALOG	ROUTINE_NAME	ROUTINE_SCHEMA
ROUTINE	ROW_COUNT	ROW_NUMBER
ROW	ROWS	
SAVEPOINT	SCALE	SCHEMA_NAME
SCHEMA	SCOPE_CATALOG	SCOPE_NAME
SCOPE_SCHEMA	SCOPE	SCROLL
SEARCH	SECOND	SECTION
SECURITY	SELECT	SELF
SENSITIVE	SEQUENCE	SERIALIZABLE
SERVER_NAME	SESSION_USER	SESSION
SET	SETS	SIMILAR
SIMPLE	SIZE	SMALLINT
SOME	SOURCE	SPACE
SPECIFIC_NAME	SPECIFIC	SPECIFICTYPE
SQL	SQLEXCEPTION	SQLSTATE
SQLWARNING	SQRT	ST_CIRCULARSTRING
ST_COMPOUNDCURVE	ST_CURVE	ST_CURVEPOLYGON
ST_GEOMCOLLECTION	ST_GEOMETRY	ST_LINESTRING

ST_MULTICURVE	ST_MULTILINESTRING	ST_MULTIPPOINT
ST_MULTIPOLYGON	ST_MULTISURFACE	ST_POINT
ST_POLYGON	ST_SURFACE	START
STATE	STATEMENT	STATIC
STDDEV_POP	STDDEV_SAMP	STRUCTURE
STYLE	SUBCLASS_ORIGIN	SUBMULTISET
SUBSTRING_REGEX	SUBSTRING	SUCCEEDS
SUM	SWITCH	SYMMETRIC
SYSTEM_TIME	SYSTEM_USER	SYSTEM
TABLE_NAME	TABLE	TABLESAMPLE
TEMPORARY	TEXT	THEN
TIES	TIME	TIMESTAMP
TIMEZONE_HOUR	TIMEZONE_MINUTE	TO
TOP_LEVEL_COUNT	TRAILING	TRANSACTION_ACTIVE
TRANSACTION	TRANSACTIONS_COMMITTED	TRANSACTIONS_ROLLED_BACK
TRANSFORM	TRANSFORMS	TRANSLATE_REGEX
TRANSLATE	TRANSLATION	TREAT
TREE	TRIGGER_CATALOG	TRIGGER_NAME
TRIGGER_SCHEMA	TRIGGER	TRIM_ARRAY
TRIM	TRUE	TRUNCATE
TYPE	TRAVERSE	
UESCAPE	UNBOUNDED	UNCOMMITTED
UNDER	UNION	UNIQUE
UNKNOWN	UNNAMED	UNNEST
UPDATE	UPPER	USAGE
USER_DEFINED_TYPE_CATALOG	USER_DEFINED_TYPE_CODE	USER_DEFINED_TYPE_NAME
USER_DEFINED_TYPE_SCHEMA	USER	USING
VALUE_OF	VALUE	VALUES
VAR_POP	VAR_SAMP	VARBINARY
VARCHAR	VARYING	VERSIONING
VERTEX	VIEW	VOID
WHEN	WHENEVER	WHERE
WHILE	WIDTH_BUCKET	WINDOW
WITH	WITHIN	WITHOUT
WORK	WRITE	
YEAR		

6.10 Restrictions for GraphScript Procedures

The table below shows the maximum allowable limit for each entry.

Description	Limit
Maximum identifier length	127
Maximum number of temporary attributes per graph	64
Maximum number of variables	10000
Maximum number of graph variables	128
Maximum script length (in bytes)	2 GB

6.11 Complex GraphScript Examples

The following example depicts a more complex example of a GraphScript procedure. It uses the Greek mythology data set and computes the number of goddesses who live in the underworld.

```
CREATE PROCEDURE "GREEK_MYTHOLOGY"."GET_NUM_OF_DAUGHTERS_IN_UNDERWORLD" (
  OUT cnt INT)
LANGUAGE GRAPH READS SQL DATA AS
BEGIN
  Graph g = Graph("GREEK_MYTHOLOGY","GRAPH");
  ALTER g ADD TEMPORARY VERTEX ATTRIBUTE(Bool livesInUnderWorld = false);
  FOREACH e IN Edges(:g) {
    Vertex src = Source(:e);
    Vertex target = Target(:e);
    Bool areGods = :src."TYPE" == 'god' AND :target."TYPE" == 'god';
    IF (:e."TYPE" == 'hasDaughter' AND :areGods) {
      IF (:target.residence == 'Underworld' AND :src.residence != 'Underworld') {
        IF (NOT :target.livesInUnderWorld) {
          cnt = :cnt + 1;
          target.livesInUnderWorld = TRUE;
        }
      }
    }
  }
END;
-- the result here is 1
-- only Persephone is a goddess and lives in the underworld
CALL "GREEK_MYTHOLOGY"."GET_NUM_OF_DAUGHTERS_IN_UNDERWORLD" (?);
```

The following example uses the Greek mythology data set and computes the number of vertices with a non-NULL residence in the 2-neighborhood of 'Chaos'. Note that this example can be simplified by omitting the

construction of the set `neighborsWithResidence`, which was included here solely to demonstrate the use of set operations.

```
CREATE PROCEDURE "GREEK_MYTHOLOGY"."NEIGHBORS_WITH_RESIDENCE" (OUT cnt BIGINT)
LANGUAGE GRAPH READS SQL DATA AS
BEGIN
  Graph g = Graph("GREEK_MYTHOLOGY", "GRAPH");
  Vertex chaos = Vertex(:g, 'Chaos');
  Multiset<Vertex> neighbors = Neighbors (:g,:chaos, 0, 2);
  Multiset<Vertex> neighborsWithResidence = Multiset<Vertex>(:g);
  FOREACH n IN :neighbors
  {
    IF (:n."RESIDENCE" IS NOT NULL)
    {
      neighborsWithResidence = :neighborsWithResidence UNION {:n};
    }
  }
  cnt = INT(COUNT(:neighborsWithResidence));
END;
CALL "GREEK_MYTHOLOGY"."NEIGHBORS_WITH_RESIDENCE" (?);
```

7 openCypher Pattern Matching

openCypher is a declarative graph query language for graph pattern matching developed by the openCypher Implementers Group (Cypher is a registered trademark of Neo4j, Inc.).

SAP HANA Graph allows you to use openCypher directly in SQL and in SAP HANA Calculation Views. This chapter describes the SQL interface and the currently supported subset of the openCypher query language. For a detailed description of how to execute an openCypher query in SAP HANA Calculation Views, see [Query Language \[page 77\]](#).

Related Information

[Query Language \[page 77\]](#)

7.1 OPENCYPHER_TABLE SQL Function

The OPENCYPHER_TABLE SQL Function enables the embedding of an openCypher query in an SQL query. The result is returned as a table. The column names and types of this table are determined by both the [openCypher query \[page 52\]](#) and the [graph workspace \[page 9\]](#).

OPENCYPHER_TABLE is very robust with respect to inconsistent graph workspaces. Errors may occur in rare cases, but usually the inconsistent parts just won't show in the result.

Syntax

```
OPENCYPHER_TABLE (  
    <graph_workspace_spec>  
    <opencypher_query_spec>  
)
```

Syntax Elements

<graph_workspace_spec>

Specifies the graph workspace used with the openCypher query.

```
<graph_workspace_spec> ::= GRAPH WORKSPACE <graph_workspace>
```

<graph_workspace> can be any graph workspace present in the system, identified by an optional schema name and a workspace name.

<opencypher_query_spec>

Specifies a query in the openCypher language.

```
<opencypher_query_spec> ::= QUERY <opencypher_query_string>
```

<opencypher_query_string> is a string or nstring literal that corresponds to a pattern matching query in the openCypher language. Although <opencypher_query_string> appears to be a normal SQL string, it is actually part of the query syntax and cannot be replaced with a string expression like most SQL strings. To parameterize it, dynamic SQL has to be used.

i Note

Since a single-quoted SQL string is used for the openCypher query, single quotes (') inside the openCypher query have to be escaped by using two single quotes (").

Examples

The example is based on the Greek mythology graph from section [Appendix B - Greek Mythology Graph Example \[page 80\]](#).

```
SELECT * FROM OPENCYPHER_TABLE( GRAPH WORKSPACE "GREEK_MYTHOLOGY"."GRAPH" QUERY
    ,
    MATCH (a)-[e]-(b)
    WHERE a.NAME = ''Hera'' RETURN b.NAME AS Name
    ORDER BY b.NAME
    ,
    )
```

Name

Ares

Cronus

Hephaestus

Rhea

Zeus

Zeus

7.2 openCypher Query Language

An openCypher query searches for subgraphs matching the specification given in MATCH clauses and returns a result table specified in the RETURN clause.

```
<openCypher_query> ::= <match_clauses> <return_clause>  
<match_clauses> ::= <match_clause> [{<match_clause>}]
```

The total number of MATCH clauses in one openCypher is currently limited to five.

7.2.1 Match Clause

The MATCH clause consists of topological constraints that define the topology (vertices, edges, and paths) of the matching subgraphs, and an optional filter condition, which allows the user to add additional (non-topological) constraints. These constraints must be fulfilled by each match in the graph workspace to become a part of the result.

```
<match_clause> ::= MATCH <topology_constraints>  
    [WHERE <non_topology_constraints>]  
<topology_constraints> ::= <topology_constraint>  
    [{<comma> <topology_constraint>}]
```

Topological Constraints

Topological constraints is a comma-separated list of vertices and edges. Additionally, one path can be used.

```
<topology_constraint> ::=  
<vertex>  
| <edge>  
| <path>
```

Edges and paths can be directed or undirected. Undirected edges and paths match edges and paths of any direction. Specification of edges and paths requires source and target vertices.

```
<vertex> ::= <l_paren> <variable_name> <r_paren>  
<edge> ::= <source> <directed_edge> <target>  
| <source> <undirected_edge> <target>  
<path> ::= <variable_name> <equal> <source> <directed_path> <target>  
| <variable_name> <equal> <source> <undirected_path> <target>  
<source> ::= <vertex>  
<target> ::= <vertex>
```

The length of paths must be provided. The minimal and maximal length of a path must be in the range 1 to 15.

```
<directed_edge> ::= <minus> <edge_var> <minus> <greater>  
<undirected_edge> ::= <minus> <edge_var> <minus>  
<directed_path> ::= <minus> <l_square> <asterisk> <uint> <double_dot> <uint>  
<r_square> <minus> <greater>  
<undirected_path> ::= <minus> <l_square> <asterisk> <uint> <double_dot> <uint>  
<r_square> <minus>
```

```
<vertex_var> ::= <l_paren> <variable_name> <r_paren>
<edge_var> ::= <l_square> <variable_name> <r_square>
```

The following examples show the use of different topological constraints in the MATCH clause:

- One vertex
MATCH (a) RETURN a.NAME AS name
- One directed edge
MATCH (a)-[e]->(b) RETURN e.TYPE AS type
- One undirected edge
MATCH (a)-[e]-(b) RETURN e.KEY AS key
- Directed variable-length path
MATCH p=(a)-[*1..2]->(b) RETURN a.NAME AS aName, b.NAME AS bName
- Undirected variable-length path
MATCH p=(a)-[*5..7]-(b) RETURN a.NAME AS aName, b.NAME AS bName

The following query illustrates the use of an undirected edge:

```
MATCH (a)-[e]-(b)
WHERE a.NAME = 'Hera' RETURN b.NAME AS Name
ORDER BY b.NAME
```

The result of this query is the following table:

Name
Ares
Cronus
Hephaestus
Rhea
Zeus
Zeus

The following query illustrates the use of a directed path:

```
MATCH p = (a)-[*1..2]->(b)
WHERE a.NAME = 'Chaos'
RETURN b.NAME AS Name
ORDER BY b.NAME
```

The path with the name p starts from vertex a and ends at vertex b. It is one or two edges long. This query returns all vertices that are related to Chaos by outgoing edges. The result is:

Name
Cronus
Gaia
Rhea
Uranus
Uranus

The following query illustrates the use of multiple MATCH clauses:

```
MATCH (a)-[e1]-(b)
MATCH (b)-[e2]-(c)
WHERE a.NAME = 'Chaos' RETURN c.NAME AS Name
ORDER BY c.NAME
```

This query returns all vertices laying at a distance of two hops from Chaos, ignoring the direction of edges:

Name
Chaos
Cronus
Gaia
Rhea
Uranus
Uranus

Note

Note that Chaos is among the results, which means that edge variables e1 and e2 match the same edge in the data graph. It would not be possible if e1 and e2 were defined in the same MATCH clause.

Non Topological Constraints

Non-topological constraints is a Boolean expression that is evaluated on each matched subgraph. This expression can contain many predicates that are combined by the logical connectives AND, OR, and NOT.

```
<non_topology_constraints> ::= <condition>
<condition> ::= <condition> OR <condition>
| <condition> AND <condition>
| NOT <condition>
| <builtin_function>
| <l_paren> <condition> <r_paren>
| <predicate>
```

Predicates include comparisons (=, <>, >, <, >=, <=), IS NULL, and other Boolean built-in functions.

```
<predicate> ::= <comparison_predicate>
| <attribute access> IS NULL
| <attribute access> CONTAINS <varchar constant>
| <attribute access> STARTS WITH <varchar constant>
| <attribute access> ENDS WITH <varchar constant>
<comparison_predicate> ::=
  <expression> <equals> <expression>
| <expression> <unequal> <expression>
| <expression> <greater> <expression>
| <expression> <smaller> <expression>
| <expression> <greater_equal> <expression>
| <expression> <smaller_equal> <expression>
<expression> ::=
  <attribute access>
| <constant>
| <string_concatenation>
```

```

<attribute access> ::= <variable_name> <dot> <attribute_name>
<variable_name> ::= <identifier>
<attribute_name> ::= <identifier>
<constant> ::= <integer_constant>
| <bigint_constant>
| <double_constant>
| <varchar_constant>
<string_concatenation> ::= <string_concatenation> <plus> <expression>
| <expression> <plus> <expression>

```

The following query illustrates string concatenation and the use of the logical connective NOT. Note that NOT has higher precedence than AND and OR and is therefore evaluated first.

```

MATCH (a)-[e]->(b)
WHERE a.NAME = 'Cro' + 'nus' AND (e.KEY < 10 OR NOT b.NAME = 'Poseidon')
RETURN b.NAME AS Name
ORDER BY b.NAME

```

The following query illustrates the use of the logical connective OR. Note that OR has lower precedence than AND.

```

MATCH (a)-[e]->(b)
WHERE a.NAME = 'Cronus' AND e.KEY < 10 OR b.NAME <> 'Poseidon'
RETURN b.NAME AS Name
ORDER BY b.NAME

```

This query returns the targets of all edges, except for two edges that are pointing at Poseidon (32 of 34 edges).

The following query illustrates the use of parentheses to evaluate the OR connective before the AND connective.

```

MATCH (a)-[e]->(b)
WHERE a.NAME = 'Cronus' AND (e.KEY < 10 OR b.NAME <> 'Poseidon')
RETURN b.NAME AS Name
ORDER BY b.NAME

```

Name
Demeter
Hades
Hera
Rhea
Zeus

The following query illustrates the use of string matching predicates for evaluating male deities.

```

MATCH (a)
WHERE a.NAME ENDS WITH 's' OR a.NAME ENDS WITH 'n'
RETURN b.NAME AS Name
ORDER BY b.NAME

```

Name
Ares
Chaos

Name

Cronus

Hades

Hephaestus

Poseidon

Uranos

Zeus

7.2.2 Return Clause

The RETURN clause lists expressions over the matched subgraphs that need to be projected into the resulting table. The optional DISTINCT specifier eliminates duplicate rows from the resulting table. The optional ORDER-BY clause allows you to sort the resulting rows in ascending or descending order. The optional LIMIT clause of the RETURN clause truncates the resulting table to the given number of rows (from the top) if the result has more rows than the given number. Otherwise all the rows are displayed. The optional SKIP clause excludes the given number of rows from the top of the result. When all three optional clauses are present, the ORDER-BY clause is applied first, then the SKIP clause, and finally the LIMIT clause.

```
reformat to:
<return_clause> ::= RETURN [ DISTINCT ] <return_list>
[<order_by_clause>] [<limit_skip_clause>]
<return_list> ::= <return_item> [{<comma> <return_item>}]
<return_item> ::= <return_expression> AS <alias>
<alias> ::= <identifier>
<return_expression ::=
<attribute_access>
| <string_concatenation>
| <aggregate_function> <l_paren> <attribute_access> <r_paren>
| <list_comprehension>
| <object_constructor>
<aggregate_function> ::= COUNT | MAX | MIN | SUM
<order_by_clause> ::= ORDER BY <order_by_list>
<order_by_list> ::= <order_by_item> [{<comma> <order_by_item>}]
<order_by_item> ::= <variable_name> <dot> <attribute_name> [ ASC | DESC ]
<list_comprehension> ::= <l_square> <variable_name> IN
<relationships_function> <pipe> <expression> <r_square>
<object_constructor> ::= <l_curly> <identifier> <colon> <expr>
[ {<comma> <identifier> <colon> <expr> } ] <r_curly>
<limit_skip_clause> ::= LIMIT <uint> [ SKIP <uint> ]
```

If there is more than one return item and at least one aggregation, the result is grouped by the not aggregated return items. The MAX, MIN and SUM functions need a numeric type as input and the result type corresponds to the input type. The COUNT function can take any type as input and the result type is always BIGINT.

An example with a complex return clause is:

```
MATCH (a)-[e]->(b)
WHERE e.TYPE = 'hasSon'
RETURN a.NAME AS name, COUNT(b.NAME) AS numSons
ORDER BY a.NAME ASC
LIMIT 5
SKIP 1
```


List comprehension is useful for extracting edge attributes of a path. List comprehension constructs a new list from the given path function returning a list of edges and the given expression that may use the given iterator variable. It is only permitted in the RETURN clause. The resulting column is a string representation of the list comprehension result in JSON format. The following expressions are allowed in the list comprehension: attribute accesses, string concatenations and constants. The following example shows how to use the list comprehension.

```
MATCH p = (a)-[*1..2]->(b)
WHERE a.NAME = 'Chaos'
RETURN [e IN RELATIONSHIPS(p) | e.TYPE] AS result
```

The result is:

result

["hasDaughter"]
["hasDaughter", "hasSon"]
["hasDaughter", "hasSon"]
["hasDaughter", "hasDaughter"]
["hasDaughter", "isMarried"]

The object constructor can group several expressions into one expression. It is only permitted in the RETURN clause. The resulting column is a string representation of the object in JSON format. The following example shows how to use the object constructor.

```
MATCH (a)-[e]->(b)
WHERE a.RESIDENCE = 'Olymp' AND b.RESIDENCE = 'Olymp' AND e.TYPE = 'marriedTo'
RETURN { from: e.SOURCE, to: e.TARGET } AS result
```

The result is:

result

{"from":"Hera","to":"Zeus"}
{"from":"Zeus","to":"Hera"}
{"from":"Aphrodite","to":"Hephaestus"}
{"from":"Hephaestus","to":"Aphrodite"}

The following openCypher query returns the values of the "NAME" attribute of all vertices in the given graph.

```
MATCH (a)
RETURN a.NAME AS name
ORDER BY a.NAME ASC
```

The result of this query is a table containing 15 rows:

Name

Aphrodite
Ares

Name

Athena

Chaos

Cronus

Demeter

Gaia

Hades

Hephaestus

Hera

Hephaestus

Hera

Persephone

Poseidon

Rhea

Uranus

Zeus

The following query illustrates the use of the LIMIT clause applied to the result from the previous example:

```
MATCH (a)
RETURN a.NAME AS name
ORDER BY a.NAME ASC
LIMIT 5
```

The result of this query is a table containing 5 rows:

Name

Aphrodite

Ares

Athena

Chaos

Cronus

The following query illustrates the use of the SKIP clause applied to the result from the previous example:

```
MATCH (a)
RETURN a.NAME AS name
ORDER BY a.NAME ASC
LIMIT 5
SKIP 1
```

The result of this query is the following table:

Name
Ares
Athena
Chaos
Cronus
Demeter

7.2.3 Keywords

All keywords are case-insensitive.

Keywords

MATCH	WHERE	OR	AND	NOT
ALL	IN	RETURN	AS	ORDER
BY	ASC	DESC	LIMIT	SKIP
WITH	CONTAINS	STARTS	ENDS	SUM
AVG	COUNT	MAX	MIN	

Reserved Keywords:

optional	distinct	unique	detach	union
unwind	merge	create	set	delete
remove	on	ascending	descending	xor
null	true	false	case	when
then	else	end	foreach	call
constraint	assert	load	csv	headers
fieldterminator	index	yield	using	drop
profile	explain	start	is	

Attribute Names, Variable Names, Aliases

Attribute names, variable names, and aliases are case-sensitive.

The following query illustrates the case sensitivity of variable names ("a" and "A" are not the same).

```
MATCH (a), (A)
RETURN a.NAME AS name
```

This query also illustrates a disconnected graph pattern, since vertices "a" and "A" are not connected by any edge. In other words, there are two connected components, each of them containing one vertex. From a performance perspective, we recommend that you avoid matching disconnected subgraphs wherever possible, because the result is the Cartesian product of matches of all connected components (225 rows), which can easily get very large and use a lot of system resources.

The following query illustrates the case sensitivity of aliases and returns the values of the "NAME" attribute of all pairs of vertices in the given graph connected by an edge.

```
MATCH (A)-[e]->(a)
RETURN a.NAME AS name, A.NAME as NAME
ORDER BY a.NAME ASC, A.NAME DESC
```

7.2.4 Built-In Functions

SAP HANA Graph supports a subset of openCypher built-in functions and offers a set of SAP HANA specific built-in functions.

```
<builtin_function> ::= <opencypher_builtin_function> | <hana_builtin_function>
```

openCypher built-in functions are case-insensitive. SAP HANA specific built-in functions are defined in the SYS namespace and are case-sensitive.

```
<hana_builtin_function> ::= SYS <dot> <text_contains_function>
<opencypher_builtin_function> ::= <all_function> | <relationships_function>
```

RELATIONSHIPS

The RELATIONSHIPS function takes a path variable and returns a collection of all edges in the path.

```
<relationships_function> ::= RELATIONSHIPS <l_paren> <variable_name> <r_paren>
```

ALL

The ALL function returns a Boolean result and can be used as a predicate in a WHERE clause. This function returns true if the given condition is true for all elements in the given collection. Otherwise it returns false.

```
<all_function> ::= ALL <l_paren> <variable_name> IN <relationships_function>
WHERE <condition> <r_paren>
```

The following example shows how to use the ALL function. If you are only interested in Chaos' daughters and daughters of daughters, you can use an additional edge filter for all edges of a path.

```
MATCH p = (a)-[*1..2]->(b)
WHERE a.NAME = 'Chaos' AND ALL(e IN RELATIONSHIPS(p) WHERE e.TYPE='hasDaughter')
RETURN b.NAME AS Name
ORDER BY b.NAME
```

The result is:

Name
Gaia
Rhea

SYS.TEXT_CONTAINS

The function SYS.TEXT_CONTAINS makes advanced text search capabilities available in openCypher.

```
<text_contains_function> ::= TEXT_CONTAINS
<l_paren> <variable_name> <dot> <attribute_name> <comma>
<varchar_constant> <comma> <varchar_constant> <r_paren>
```

This function returns a Boolean result and can be used as a predicate in a WHERE clause of an openCypher query.

The first parameter is a vertex or directed edge attribute of types VARCHAR or NVARCHAR. The second parameter is a string literal that specifies the search pattern, and the third parameter is a string literal specifying one of the three available search modes: EXACT, LINGUISTIC, or FUZZY. The search pattern might contain reserved operators (-,?,",*,OR,%) that have special meaning, depending on the search mode.

Here are two examples of how to apply the TEXT_CONTAINS function:

```
MATCH (a)
WHERE SYS.TEXT_CONTAINS(a.name, 'Philip', 'FUZZY(0.8)')
RETURN a.id
```

```
MATCH (a)-[e]->(b)
WHERE SYS.TEXT_CONTAINS(e.color, 'b*', 'EXACT')
RETURN a.id
```

Moreover, the user can create a full-text index for any VARCHAR or NVARCHAR column in the vertex or edge table to transform the search into a full-text search. More information about the full-text search and how to create a full-text index can be found in the [SAP HANA Search Developer Guide](#) and in the [SAP HANA SQL Reference Guide](#).

There are the following limitations:

- Undirected edges are not supported.
- The function only supports graph workspaces on top of column store tables.
- The search pattern must not contain single quotes or brackets.

Exact search

In the EXACT search mode the search term must match the entire search pattern to return a column entry as a match. In this search mode, the user can specify the search pattern as a complex predicate using the following specific operators. Some operators are useful only in a full-text search.

- With a minus sign (-), SAP HANA searches in columns for matches that do not contain the term immediately following the minus sign.
- The question mark (?) replaces a single character in a search term (for example, `cat?` would match `cats`).
- Terms within the quotation marks (" ") are not tokenized and are handled as a string. Therefore, all search matches must be exact.
- The or-operator (OR) matches contain at least one of the terms joined by the OR operator.
- The asterisk sign (*) replaces 0 or more characters in a search term (for example, `cat*` would match `cats` and `catalogs`).
- The space operator () matches contain both of the terms joined by a space operator.

Examples for complex search patterns in the EXACT search mode are:

```
SYS.TEXT_CONTAINS(a.color, 'red', 'EXACT')
SYS.TEXT_CONTAINS(a.color, '-red', 'EXACT')
SYS.TEXT_CONTAINS(a.color, 'red*', 'EXACT')
SYS.TEXT_CONTAINS(a.color, '?ed', 'EXACT')
SYS.TEXT_CONTAINS(a.color, 'red OR blue', 'EXACT')
SYS.TEXT_CONTAINS(a.color, 'red blue', 'EXACT')
SYS.TEXT_CONTAINS(a.color, '"red blue"', 'EXACT')
```

Linguistic search

A linguistic search finds all words that have the same word stem as the search term. It also finds all words for which the search term is the word stem. When you execute a linguistic search, the system has to determine the stems of the searched terms. It will look up the stems in the stem dictionary. The hits in the stem dictionary point to all words in the word dictionary that have this stem.

```
SYS.TEXT_CONTAINS(a.attribute, 'produced', 'LINGUISTIC')
```

A linguistic search for 'produced' will also find 'producing' and 'produce'.

Fuzzy search

Fuzzy search is a fast and fault-tolerant search feature in SAP HANA. A fuzzy search returns records even if the search term contains additional or missing characters or other types of spelling errors. The first parameter, the fuzzy score, defines how different the search pattern can be. It is a floating point number between 0 and 1 and the larger the number, the less difference is allowed. Moreover, the fuzzy search mode takes many additional parameters that define how the fuzzy score is calculated.

```
SYS.TEXT_CONTAINS(a.name, 'Philip', 'FUZZY(0.8)')
SYS.TEXT_CONTAINS(a.name, 'Philip', 'FUZZY(0.8, option1=value1, option2=value2)')
```

A fuzzy search for 'Philip' with an appropriate fuzzy score also returns vertices with the name 'Philipp'. This provides the user the possibility to query dirty data, for example, querying misspelled names.

The following table shows additional options to influence fuzzy search results. Some options are only usable with or without a full-text index and many options conflict with each other or need to be combined with different options. For more detailed information about these options, see the [SAP HANA Search Developer Guide](#).

OPTIONS	VALUES	DESCRIPTION
similarCalculationMode	typeahead, symmetricsearch, substringsearch, searchcompare, search, compare	Defines the impact of wrong characters, additional characters in search pattern, and additional characters in data for the fuzzy score
termMappingTable	unquoted sql identifier	Defines terms that are used to extend a search to generate additional results
stopWordTable	unquoted sql identifier	Defines terms that are less significant for a search and are therefore not used to generate results
abbreviationSimilarity	0..1	Defines the similarity that is returned for a matching initial character
andSymmetric	true, false	Symmetric content search for 'ABC' finds 'ABCD' and 'AB'
andThreshold	0..1	Determines the percentage of tokens that need to match
(de)composeWords	1..5	Control sensitivity regarding compound spelling
searchMode	alphanumeric, housenumber, postcode	Search for special formats

Related Information

[SAP HANA Search Developer Guide](#)
[SAP HANA SQL and System Views Reference](#)

7.2.5 Basic Building Blocks

Basic Rules

```

<identifier> ::= <simple_identifier> | <special_identifier>
<simple_identifier> ::= (<letter> | <underscore>)
    {<letter> | <underscore> | <digit>}
<special_identifier> ::=
<backtick> { (<any_char> - <backtick>) | <backtick><backtick> }<backtick>
<integer_constant>::
<uint>
| <minus> <uint>
<bigint_constant> ::= <uint>
| <minus> <uint>

```

```

<double_constant> ::= ( <zero> | <pos_digit> {<digit>} ) <dot> <digit> [{<digit>}]
| <minus> ( <zero> | <pos_digit> {<digit>} ) <dot> <digit> [{<digit>}]
<uint> ::= <zero>
| <positive_digit> [{<digit>}]
<varchar_constant> ::= <single_quote> { ( <any_char> -
( <single_quote> | <backslash> ) ) | <escaped_char> } <single_quote>
<escaped_character> ::= <backslash> <single_quote>
| <backslash> <backslash>

```

BNF Lowest Terms Representations

```

<backtick> ::= `
<dot> ::= .
<double_dot> ::= ..
<comma> ::= ,
<minus> ::= -
<l_paren> ::= (
<r_paren> ::= )
<l_curly> ::= {
<r_curly> ::= }
<l_square> ::= [
<r_square> ::= ]
<greater> ::= >
<equal> ::= =
<lower> ::= <
<lower_equal> ::= <=
<greater> ::= >
<greater_equal> ::= >=
<unequal> ::= <>
<asterisk> ::= *
<single_quote> ::= '
<double_quote> ::= "
<backslash> ::= \
<zero> ::= 0
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<positive_digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<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 | 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
<any_char> ::= any character !!

```


8 Graph Algorithms

SAP HANA Graph provides a graph calculation node that can be used in calculation scenarios.

This node allows you to execute one of the available actions on the given graph workspace and provide results as table output.

Calculation scenarios can be created with plain SQL as shown in the following section or with tools such as the SAP HANA Modeler (see [SAP HANA Modeling Guide](#)) or a native SAP HANA Graph Viewer (see [Appendix A – SAP HANA Graph Viewer \[page 79\]](#)).

A graph node has the following parameters:

Parameter	Value
schema	Graph workspace schema (for example "GREEK_MYTHOLOGY")
workspace	Graph workspace name (for example "GRAPH")
action	One of the following actions: GET_NEIGHBORHOOD, GET_SHORTEST_PATHS_ONE_TO_ALL, GET_SHORTEST_PATH_ONE_TO_ONE, GET_STRONGLY_CONNECTED_COMPONENTS, MATCH_SUBGRAPHS

In case of an inconsistent graph workspace, the calculation scenario will exit with an error.

To use the graph node, a user needs SELECT privileges on the given graph workspace.

Each action has a set of additional parameters. The remainder of this section describes the available actions and their additional parameters.

Related Information

[SAP HANA Modeling Guide \(SAP HANA XS advanced model\)](#)

[SAP HANA Modeling Guide \(SAP HANA web workbench\)](#)

[SAP HANA Modeling Guide \(SAP HANA studio\)](#)

8.1 Neighborhood Search (Breadth-First Search)

The GET_NEIGHBORHOOD graph action retrieves the neighboring vertices within the given radius (depth) from the given start vertices.

This action allows you to specify multiple start vertices, choose traversal direction, setting filters on vertices and edges, and setting minimum and maximum depth (radius) of the neighborhood.

This action has the following additional parameters:

Parameter	Value
startVertices	Set of start vertex keys
direction	Traversal direction, can use one of the following values: any , incoming , outgoing
minDepth	Minimum depth (radius) of the neighborhood. 0 means the start vertices are included into the result.
maxDepth	Maximum depth (radius) of the neighborhood.
vertexFilter	Vertex filter expression comparable to SQL's where-clause on the vertex table (default: empty)
edgeFilter	Edge filter expression comparable to SQL's where-clause on the edge table (default: empty)
depthColumn	Depth column name (default: "DEPTH")

The output includes the following:

Parameter	Value
vertex table	Contains the set of explored vertices with vertex key and the depth level of the neighborhood.

The SQL statement used in the following code sample creates a calculation scenario with a single graph node with the GET_NEIGHBORHOOD action on the graph workspace "GREEK_MYTHOLOGY"."GRAPH". This scenario traverses the underlying graph using all outgoing edges starting from vertex 'Chaos' and returns vertices with minimum depth 0 and maximum depth 2 from the start vertex.

```
CREATE CALCULATION SCENARIO "GREEK_MYTHOLOGY"."GET_NEIGHBORHOOD_EXAMPLE" USING '
<?xml version="1.0"?>
<cubeSchema version="2" operation="createCalculationScenario"
  defaultLanguage="en">
  <calculationScenario schema="GREEK_MYTHOLOGY" name="GET_NEIGHBORHOOD_EXAMPLE">
    <calculationViews>
      <graph name="get_neighborhood_node" defaultViewFlag="true"
        schema="GREEK_MYTHOLOGY" workspace="GRAPH" action="GET_NEIGHBORHOOD">
        <expression>
          <![CDATA[{
            "parameters": {
              "startVertices": ["Chaos"],
              "direction": "outgoing",
              "minDepth": 0,
              "maxDepth": 2
            }
          }]]>
        </expression>
        <viewAttributes>
          <viewAttribute name="NAME" datatype="string"/>
          <viewAttribute name="DEPTH" datatype="int"/>
        </viewAttributes>
      </graph>
    </calculationViews>
  </calculationScenario>
</cubeSchema>
' WITH PARAMETERS
  ('EXPOSE_NODE'=('get_neighborhood_node', 'GET_NEIGHBORHOOD_EXAMPLE'));
```

The following SQL statement executes the calculation scenario and orders the resulting vertices by depth in reverse order.

```
SELECT * FROM "GREEK_MYTHOLOGY"."GET_NEIGHBORHOOD_EXAMPLE" ORDER BY "DEPTH";
```

The result of this operation is the following vertex table:

NAME	DEPTH
Chaos	0
Gaia	1
Uranus	2
Cronus	2
Rhea	2

The same result can be obtained through the SAP HANA Graph Viewer web tool (see [Appendix A – SAP HANA Graph Viewer \[page 79\]](#)).

The following SQL statement deletes the calculation scenario and its corresponding view:

```
DROP CALCULATION SCENARIO "GREEK_MYTHOLOGY"."GET_NEIGHBORHOOD_EXAMPLE" CASCADE;
```

i Note

The SAP HANA Graph Viewer can be used to create the calculation scenario, execute the neighborhood search on the selected graph workspace with specific parameters and to visualize the result. Afterwards the generated calculation scenario can be reused in all kinds of SQL queries.

8.2 Shortest Path

These actions provide the information for the shortest path.

8.2.1 Shortest Path (One-to-All)

The action `GET_SHORTEST_PATHS_ONE_TO_ALL` returns the shortest paths from the provided start vertex to all reachable vertices in the graph - also known as single-source shortest path (SSSP). The resulting shortest paths form a tree structure with the start vertex at the root. All other vertices carry the shortest distance (smallest weight) information. The non-negative edge weights are read from the column provided in the edge table.

This action has the following additional parameters:

Parameter	Need	Value
startVertex	mandatory	Start vertex key
inputWeightColumn	optional	A column in the edge table that contains edge weights. If omitted, edge weights are set to 1.
outputWeightColumn	optional	Output weight (shortest distance) column name (default: "WEIGHT")
sourceColumn	optional	Name in output table that contains the source of the traversed edge; Default: source column name of graph workspace
targetColumn	optional	Name in output table that contains the target of the traversed edge; Default: target column name of graph workspace
direction	optional	Traversal direction, can use one of the following values: any, incoming, outgoing

The output consists of the following:

Parameter	Value
vertex table	Contains vertex keys and corresponding smallest weights (shortest distances)
edge table	Optional edge table with shortest path(s)

The SQL statement used in the following code sample creates a calculation scenario with a single graph node with the action GET_SHORTEST_PATHS_ONE_TO_ALL on the workspace "GREEK_MYTHOLOGY"."GRAPH". This scenario calculates shortest paths from the vertex 'Chaos' to all other vertices. Since input weight column parameter is not specified, the weight of each edge is considered as 1.

```
CREATE CALCULATION SCENARIO "GREEK_MYTHOLOGY"."SSSP_EXAMPLE" USING '
<?xml version="1.0"?>
<cubeSchema version="2" operation="createCalculationScenario"
  defaultLanguage="en">
  <calculationScenario schema="GREEK_MYTHOLOGY" name="SSSP_EXAMPLE">
    <calculationViews>
      <graph name="sssp_node" defaultViewFlag="true" schema="GREEK_MYTHOLOGY"
        workspace="GRAPH" action="GET_SHORTEST_PATHS_ONE_TO_ALL">
        <expression>
          <![CDATA[{
            "parameters": {
              "startVertex": "Chaos",
              "outputWeightColumn": "DISTANCE"
            }
          }]]>
        </expression>
        <viewAttributes>
          <viewAttribute name="NAME" datatype="string"/>
          <viewAttribute name="DISTANCE" datatype="int"/>
        </viewAttributes>
      </graph>
```

```

    </calculationViews>
  </calculationScenario>
</cubeSchema>
' WITH PARAMETERS ('EXPOSE_NODE'=('sssp_node', 'SSSP_EXAMPLE'));

```

The following SQL statement executes the calculation scenario and sorts the result by the output weight column "DISTANCE".

```
SELECT * FROM "GREEK_MYTHOLOGY"."SSSP_EXAMPLE" ORDER BY "DISTANCE";
```

The result of this operation is the vertex table with an additional column for shortest distance:

NAME	DISTANCE
Chaos	0
Gaia	1
Uranus	2
Cronus	2
Rhea	2
Zeus	3
Poseidon	3
Hades	3
Aphrodite	3
Demeter	3
Hera	3
Ares	4
Athena	4
Hephaestus	4
Persephone	4

Note

The same results can also be obtained by executing GET_NEIGHBORHOOD action with startVertices equal to 'Chaos', direction equal to 'outgoing', minDepth equal to '0', maxDepth equal to '*', and depthColumn equal to 'DISTANCE'.

The following SQL statement drops the calculation scenario and all its views.

```
DROP CALCULATION SCENARIO "GREEK_MYTHOLOGY"."SSSP_EXAMPLE" CASCADE;
```

The following statement creates a calculation scenario that returns both vertices and edges of shortest paths from the start vertex 'Chaos' to all other vertices in the graph workspace "GREEK_MYTHOLOGY"."GRAPH".

```

CREATE CALCULATION SCENARIO "GREEK_MYTHOLOGY"."SSSP2_EXAMPLE" USING '
<?xml version="1.0"?>
<cubeSchema version="3" operation="createCalculationScenario">
  <calculationScenario schema="GREEK_MYTHOLOGY" name="SSSP2_EXAMPLE">
    <calculationViews>
      <multipleOutputGraph name="sssp2_node" defaultViewFlag="false"

```

```

schema="GREEK_MYTHOLOGY" workspace="GRAPH"
action="GET_SHORTEST_PATHS_ONE_TO_ALL">
  <expression>
    <![CDATA[{
      "parameters": {
        "startVertex": "Chaos",
        "outputWeightColumn": "DISTANCE"
      }
    }]]>
  </expression>
  <outputs>
    <output name="vertices">
      <attributes>
        <attribute name="NAME" datatype="string"/>
        <attribute name="DISTANCE" datatype="int"/>
      </attributes>
    </output>
    <output name="edges">
      <attributes>
        <attribute name="KEY" datatype="int"/>
        <attribute name="SOURCE" datatype="string"/>
        <attribute name="TARGET" datatype="string"/>
      </attributes>
    </output>
  </outputs>
</multipleOutputGraph>
<projection name="projectVertices" defaultViewFlag="true">
  <inputs>
    <input name="sssp2_node" outputName="vertices"/>
  </inputs>
  <attributes>
    <attribute name="NAME" datatype="string"/>
    <attribute name="DISTANCE" datatype="int"/>
  </attributes>
</projection>
<projection name="projectEdges" defaultViewFlag="false">
  <inputs>
    <input name="sssp2_node" outputName="edges"/>
  </inputs>
  <attributes>
    <attribute name="KEY" datatype="int"/>
    <attribute name="SOURCE" datatype="string"/>
    <attribute name="TARGET" datatype="string"/>
  </attributes>
</projection>
</calculationViews>
</calculationScenario>
</cubeSchema>
' WITH PARAMETERS ('EXPOSE_NODE'=('projectVertices', 'SSSP2_EXAMPLE'));
CREATE COLUMN VIEW "GREEK_MYTHOLOGY"."SSSP2_EXAMPLE_EDGES" WITH PARAMETERS (
  indexType=11,
  'PARENTCALCINDEXPACKAGE'='GREEK_MYTHOLOGY',
  'PARENTCALCINDEX'='SSSP2_EXAMPLE',
  'PARENTCALCNODE'='projectEdges');

```

The following SQL statement executes the calculation scenario and returns edges of all shortest paths ordered by edge keys.

```
SELECT * FROM "GREEK_MYTHOLOGY"."SSSP2_EXAMPLE_EDGES" ORDER BY "KEY";
```

The result of this operation is the edge table containing edge keys, source vertex keys and target vertex keys:

KEY	SOURCE	TARGET
1	Chaos	Gaia
2	Gaia	Uranus
3	Gaia	Cronus
5	Gaia	Rhea
7	Cronus	Zeus
9	Cronus	Hera
11	Cronus	Demeter
13	Cronus	Poseidon
15	Cronus	Hades
17	Zeus	Athena
18	Zeus	Ares
20	Uranus	Aphrodite
21	Zeus	Hephaestus
23	Zeus	Persephone

The following SQL statement drops the calculation scenario and all its views.

```
DROP CALCULATION SCENARIO "GREEK_MYTHOLOGY"."SSSP2_EXAMPLE" CASCADE;
```

Note

The SAP HANA Graph Viewer can be used to create the calculation scenario, execute the shortest path algorithm on the selected graph workspace with specific parameters and to visualize the result. Afterwards the generated calculation scenario can be reused in all kinds of SQL queries.

8.2.2 Shortest Path (One-to-One)

The action GET_SHORTEST_PATH_ONE_TO_ONE returns the shortest path from the provided start vertex to the provided target vertex - also known as single-source single-target shortest path (SSSTSP).

The resulting table contains all vertices of the shortest path between the start vertex and the target vertex with the distance from the start vertex.

If there is more than one shortest path between the two vertices, only one of them is returned.

The non-negative edge weights are read from the column provided in the edge table.

Parameter	Need	Value
startVertex	mandatory	Start vertex key

Parameter	Need	Value
targetVertex	mandatory	Target vertex key
inputWeightColumn	optional	A column in the edge table that contains edge weights. If omitted, edge weights are set to 1.
outputWeightColumn	optional	Output weight (shortest distance) column name (default: "WEIGHT")
sourceColumn	optional	Name in output table that contains the source of the traversed edge. Default: source column name of graph workspace
targetColumn	optional	Name in output table that contains the target of the traversed edge. Default: target column name of graph workspace
Direction	optional	Traversal direction, can use one of the following values: any, incoming, outgoing
orderingColumn	optional	Name of the column that contains the unweighted distance. Default: ORDERING

Examples

```

DROP CALCULATION SCENARIO "GREEK_FAMILY"."SSSTSP_EXAMPLE" cascade;
CREATE CALCULATION SCENARIO "GREEK_FAMILY"."SSSTSP_EXAMPLE" USING '
<?xml version="1.0"?>
<cubeSchema version="2" operation="createCalculationScenario"
  defaultLanguage="en">
  <calculationScenario schema="GREEK_FAMILY" name="SSSTSP_EXAMPLE">
  <calculationViews>
  <graph name="ssstsp_node" defaultViewFlag="true" schema="GREEK_FAMILY"
    workspace="GRAPH" action="GET_SHORTEST_PATH_ONE_TO_ONE">
  <expression>
  <![CDATA[{
  "parameters":
  {
  "startVertex":
  "Chaos",
  "targetVertex":
  "Zeus",
  "outputWeightColumn":
  "DISTANCE"
  }
  }]]>
  </expression>
  <viewAttributes>
  <viewAttribute name="ORDERING" datatype="Fixed" length="18" sqltype="BIGINT"/>
  <viewAttribute name="SOURCE" datatype="string"/>
  <viewAttribute name="TARGET" datatype="string"/>
  <viewAttribute name="DISTANCE" datatype="int"/>
  </viewAttributes>
  </graph>

```



```

</calculationViews>
</calculationScenario>
</cubeSchema>
' WITH PARAMETERS ('EXPOSE_NODE'=('ssstsp_node', 'SSSTSP_EXAMPLE'));

```

The following SQL statement executes the calculation scenario and sorts the result by the column "ORDERING":

```
select * from "GREEK_FAMILY"."SSSTSP_EXAMPLE" order by ORDERING;
```

The result of this operation is the following table:

ORDERING	SOURCE	TARGET	DISTANCE
1	Chaos	Gaia	1
2	Gaia	Rhea	2
3	Rhea	Zeus	3

8.3 Strongly Connected Components

Action `GET_STRONGLY_CONNECTED_COMPONENTS` calculates strongly connected components (SCC) in the given graph workspace.

A directed graph is said to be strongly connected if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected.

The only output of this action is a table containing the vertex key column and a "COMPONENT" column containing strongly connected component indices. All vertices with the same component index belong to the same strongly connected component.

This action has the following additional parameters:

Parameter	Value
componentColumn	Component index column name (default: "COMPONENT")

The output consists of the following:

Parameter	Value
vertex table	Contains vertex key column and component index column

The SQL statement used in the following code sample creates a calculation scenario with a single graph node with the `GET_STRONGLY_CONNECTED_COMPONENTS` action on the graph workspace "GREEK_MYTHOLOGY"."GRAPH".

```

CREATE CALCULATION SCENARIO "GREEK_MYTHOLOGY"."SCC_EXAMPLE" USING '
<?xml version="1.0"?>
<cubeSchema version="2" operation="createCalculationScenario"
  defaultLanguage="en">
  <calculationScenario schema="GREEK_MYTHOLOGY" name="SCC_EXAMPLE">

```

```

<calculationViews>
  <graph name="scc_node" defaultViewFlag="true" schema="GREEK_MYTHOLOGY"
workspace="GRAPH" action="GET_STRONGLY_CONNECTED_COMPONENTS">
    <expression>
    </expression>
    <viewAttributes>
      <viewAttribute name="NAME" datatype="string"/>
      <viewAttribute name="COMPONENT" datatype="int"/>
    </viewAttributes>
    </graph>
  </calculationViews>
</calculationScenario>
</cubeSchema>
' WITH PARAMETERS ('EXPOSE_NODE'=('scc_node', 'SCC_EXAMPLE'));

```

The following SQL statement executes the calculation scenario and sorts the result by the component index.

```
SELECT * FROM "GREEK_MYTHOLOGY"."SCC_EXAMPLE" ORDER BY "COMPONENT";
```

The result of this statement is the following table.

NAME	COMPONENT
Athena	1
Ares	2
Aphrodite	3
Hephaestus	3
Hades	4
Persephone	4
Zeus	5
Hera	5
Demeter	6
Poseidon	7
Rhea	8
Cronus	8
Gaia	9
Uranus	9
Chaos	10

The same result can be obtained through the SAP HANA Graph Viewer web tool (see [Appendix A – SAP HANA Graph Viewer \[page 79\]](#)).

In our example, the fifteen vertices are partitioned into ten strongly connected components, because there are five couples and the 'marriedTo' relationship goes in both directions, forming a cycle of two vertices.

The following SQL statement deletes the calculation scenario and its corresponding view.

```
DROP CALCULATION SCENARIO "GREEK_MYTHOLOGY"."SCC_EXAMPLE" CASCADE;
```

8.4 Graph Algorithm Variables

Graph variables allow to parameterize calculation scenarios, so that the same graph action can be called with different parameters.

Using the graph variables, a user needs to create only one calculation scenario for a combination of the graph workspace and the graph action.

The following SQL statement illustrates a parameterized calculation scenario for the GET_NEIGHBORHOOD graph action and the "GREEK_MYTHOLOGY"."GRAPH" graph workspace that uses graph variables.

```
CREATE CALCULATION SCENARIO "GREEK_MYTHOLOGY"."GET_NEIGHBORHOOD_EXAMPLE2"
USING '
<?xml version="1.0"?>
<cubeSchema version="2" operation="createCalculationScenario"
  defaultLanguage="en">
  <calculationScenario schema="GREEK_MYTHOLOGY"
    name="GET_NEIGHBORHOOD_EXAMPLE2">
    <calculationViews>
      <graph name="get_neighborhood_node" defaultViewFlag="true"
        schema="GREEK_MYTHOLOGY" workspace="GRAPH" action="GET_NEIGHBORHOOD">
        <expression>
          <![CDATA[{
            "parameters": {
              "startVertices": $$startVertices$$,
              "direction": "$$direction$$",
              "minDepth": $$minDepth$$,
              "maxDepth": $$maxDepth$$,
              "vertexFilter" : "$$vertexFilter$$",
              "edgeFilter" : "$$edgeFilter$$"
            }
          }]]>
        </expression>
        <viewAttributes>
          <viewAttribute name="NAME" datatype="string"/>
          <viewAttribute name="DEPTH" datatype="int"/>
        </viewAttributes>
      </graph>
    </calculationViews>
    <variables>
      <variable name="$$startVertices$$" type="graphVariable"/>
      <variable name="$$direction$$" type="graphVariable">
        <defaultValue>outgoing</defaultValue>
      </variable>
      <variable name="$$minDepth$$" type="graphVariable">
        <defaultValue>0</defaultValue>
      </variable>
      <variable name="$$maxDepth$$" type="graphVariable"/>
      <variable name="$$vertexFilter$$" type="graphVariable">
        <defaultValue></defaultValue>
      </variable>
      <variable name="$$edgeFilter$$" type="graphVariable">
        <defaultValue></defaultValue>
      </variable>
    </variables>
  </calculationScenario>
</cubeSchema>
' WITH PARAMETERS (
  'EXPOSE_NODE'=('get_neighborhood_node',
  'GET_NEIGHBORHOOD_EXAMPLE2');
```

The following SQL statement executes the calculation scenario

"GREEK_MYTHOLOGY"."GET_NEIGHBORHOOD_EXAMPLE2" with the specific parameter values.

```
SELECT * FROM "GREEK_MYTHOLOGY&quot;"GET_NEIGHBORHOOD_EXAMPLE2"  
(placeholder."$$startVertices$$" => '["Chaos"]',  
placeholder."$$direction$$" => 'outgoing',  
placeholder."$$minDepth$$" => '0',  
placeholder."$$maxDepth$$" => '2',  
placeholder."$$vertexFilter$$" => '',  
placeholder."$$edgeFilter$$" => '');
```

The graph variables `$$direction$$`, `$$minDepth$$`, `$$vertexFilter$$` and `$$edgeFilter$$` all have defined default values and therefore can be omitted when executing the calculation scenario.

```
SELECT * FROM "GREEK_MYTHOLOGY"."GET_NEIGHBORHOOD_EXAMPLE2" ORDER BY "DEPTH"  
WITH PARAMETERS (  
  'placeholder' = ('$$startVertices$$', '["Chaos"]'),  
  'placeholder' = ('$$maxDepth$$', '2')  
);
```

i Note

The SAP HANA Graph Viewer can be used to create the calculation scenario, execute the strongly connected component algorithm on the selected graph workspace with specific parameters, and visualize the result. Afterwards the generated calculation scenario can be reused in all kinds of SQL queries.

8.5 Pattern Matching

Pattern matching is a kind of graph query, which involves finding all the subgraphs (within a graph) that match the given pattern.

SAP HANA Graph provides two options for executing graph pattern queries. The first option is to use the graphical pattern editor in SAP Web IDE for SAP HANA. The second option is to describe the pattern in openCypher query language.

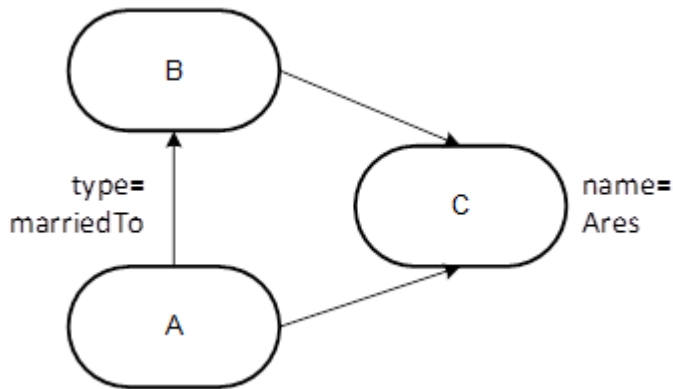
8.5.1 Graphical Pattern Editor

The SAP Web IDE for SAP HANA offers a graphical editor for composing pattern matching queries.

A graphical pattern contains a set of vertex variables, a set of edge variables, a set of filter conditions, a projection list, an order-by list, a limit, and an offset.

The result of a graphical pattern matching query is a projection of subgraphs within a given graph workspace that match the given pattern. The columns of the result table correspond to the projection list of the given pattern. Every row in the result table corresponds to a matching subgraph.

The following example represents a subgraph pattern that contains three vertices and three edges. The first two vertices A and B are connected with an edge filtered by attribute `TYPE` of value `marriedTo`. The third vertex C is connected to the first two vertices through incoming edges and vertex C is filtered by attribute `NAME` on value `Ares`. The pattern searches for parents of god Ares.



When projecting on the key column of each edge, the following table shows the result:

A	B	C
Zeus	Hera	Ares
Hera	Zeus	Ares

i Note

We get the same matching subgraph twice because our pattern has a symmetry (or two automorphisms, to be precise). In other words, we do not specify which of the two is the father and which is the mother in our data, and therefore both options are valid answers (see Appendix B: Greek Mythology Graph Example). The symmetry in this particular query can be avoided by adding an extra condition on A and B vertices: `A.NAME < B.NAME`.

8.5.2 Query Language

The action `MATCH_SUBGRAPHS` allows you to execute pattern matching queries written in the supported subset of the [openCypher Query Language \[page 52\]](#) on the given graph workspace.

i Note

openCypher queries provide different results than graphical pattern matching. In graphical pattern matching, the uniqueness of edges and vertices is enforced. In contrast, the semantic of an openCypher query only enforces the uniqueness of edges.

expression

A query string written in the supported subset of the openCypher query language containing one `MATCH` clause and one `RETURN` clause.

The output consists of the following:

result table

Every row corresponds to a matching subgraph. Every row contains attributes of vertices and edges specified in the RETURN clause.

The SQL statement used in the following code sample creates a calculation scenario with a single graph node with the MATCH_SUBGRAPHS action on the graph workspace "GREEK_MYTHOLOGY"."GRAPH". The openCypher query searches for the married parents of the Greek god Ares. Note that single quotes inside the query need to be properly escaped when creating a calculation scenario.

```
CREATE CALCULATION SCENARIO "GREEK_MYTHOLOGY"."MATCH_SUBGRAPHS_EXAMPLE" USING '
<?xml version="1.0"?>
<cubeSchema version="2" operation="createCalculationScenario"
  defaultLanguage="en">
  <calculationScenario schema="GREEK_MYTHOLOGY" name="MATCH_SUBGRAPHS_EXAMPLE">
  <calculationViews>
  <graph name="match_subgraphs_node" defaultViewFlag="true"
    schema="GREEK_MYTHOLOGY" workspace="GRAPH" action="MATCH_SUBGRAPHS">
  <expression>
  <![CDATA[
MATCH (A)-[E1]->(B), (A)-[E2]->(C), (B)-[E3]->(C) WHERE E1.TYPE = 'marriedTo'
AND C.NAME = 'Ares' RETURN A.NAME AS PARENT1_NAME, B.NAME AS PARENT2_NAME
]]>
  </expression>
  <viewAttributes>
  <viewAttribute name="PARENT1_NAME" datatype="string"/>
  <viewAttribute name="PARENT2_NAME" datatype="string"/>
  </viewAttributes>
  </graph>
  </calculationViews>
  </calculationScenario>
</cubeSchema>' WITH PARAMETERS
('EXPOSE_NODE'=('match_subgraphs_node', 'MATCH_SUBGRAPHS_EXAMPLE'));
```

The following SQL statement executes the calculation scenario and orders the results by the first parent's name and then by the second parent's name.

```
SELECT "PARENT1_NAME", "PARENT2_NAME" FROM
  "GREEK_MYTHOLOGY"."MATCH_SUBGRAPHS_EXAMPLE"
ORDER BY "PARENT1_NAME", "PARENT2_NAME";
```

9 Additional Information

This section provides additional information.

9.1 Appendix A – SAP HANA Graph Viewer

The Graph Viewer is a native SAP HANA application that provides an interface to interact with and visualize graph workspaces in SAP HANA.

9.1.1 Install SAP HANA Graph Viewer

SAP HANA Graph Viewer is an additional tool for SAP HANA Graph that can be downloaded from the [SAP Software Downloads](#).

1. Open the SAP Software Downloads.
2. Search for "SAP HANA Graph Viewer".
3. Download the HCOGRAPHVIEWER<nn_n>-<mmm>.ZIP file and unpack it.
4. Save the HCOGRAPHVIEWER.tgz file on your client.
5. Open the SAP HANA studio.
6. Choose File > Import.
7. Choose SAP HANA Content > Delivery Unit.
8. Select Client.
9. Browse to the file HCOGRAPHVIEWER.tgz, select it, and choose Finish.

For more information, see [SAP Note 2306732 - SAP HANA Graph Viewer](#) and *Deploy a Delivery Unit Archive (*.tgz)* in the [SAP HANA Administration Guide](#).

i Note

After the installation, the SAP HANA Graph Viewer can be accessed via the URL `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/graph/viewer/`

The SAP HANA Graph Viewer supports the following Web browsers: *Chrome* and *Mozilla Firefox*.

Related Information

[SAP Note 2306732](#)

[SAP Software Downloads](#)

9.2 Appendix B - Greek Mythology Graph Example

You can use the following SQL statements to create the Greek mythology graph example.

CREATE Vertex Table MEMBERS and Edge Table RELATIONSHIPS

```
CREATE SCHEMA "GREEK_MYTHOLOGY";
CREATE COLUMN TABLE "GREEK_MYTHOLOGY"."MEMBERS" (
  "NAME" VARCHAR(100) PRIMARY KEY,
  "TYPE" VARCHAR(100),
  "RESIDENCE" VARCHAR(100)
);
CREATE COLUMN TABLE "GREEK_MYTHOLOGY"."RELATIONSHIPS" (
  "KEY" INT UNIQUE NOT NULL,
  "SOURCE" VARCHAR(100) NOT NULL
  REFERENCES "GREEK_MYTHOLOGY"."MEMBERS" ("NAME")
  ON UPDATE CASCADE ON DELETE CASCADE,
  "TARGET" VARCHAR(100) NOT NULL
  REFERENCES "GREEK_MYTHOLOGY"."MEMBERS" ("NAME")
  ON UPDATE CASCADE ON DELETE CASCADE,
  "TYPE" VARCHAR(100)
);
```

CREATE Graph Workspace GRAPH

```
CREATE GRAPH WORKSPACE "GREEK_MYTHOLOGY"."GRAPH"
EDGE TABLE "GREEK_MYTHOLOGY"."RELATIONSHIPS"
SOURCE COLUMN "SOURCE"
TARGET COLUMN "TARGET"
KEY COLUMN "KEY"
VERTEX TABLE "GREEK_MYTHOLOGY"."MEMBERS"
KEY COLUMN "NAME";
```

INSERT Rows into Vertex Table MEMBERS

```
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE")
VALUES ('Chaos', 'primordial deity');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE")
VALUES ('Gaia', 'primordial deity');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE")
VALUES ('Uranus', 'primordial deity');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
```



```

VALUES ('Rhea', 'titan', 'Tartarus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Cronus', 'titan', 'Tartarus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Zeus', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Poseidon', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Hades', 'god', 'Underworld');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Hera', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Demeter', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Athena', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Ares', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Aphrodite', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Hephaestus', 'god', 'Olympus');
INSERT INTO "GREEK_MYTHOLOGY"."MEMBERS"("NAME", "TYPE", "RESIDENCE")
VALUES ('Persephone', 'god', 'Underworld');

```

INSERT Rows into Edge Table RELATIONSHIPS

```

INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (1, 'Chaos', 'Gaia', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (2, 'Gaia', 'Uranus', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (3, 'Gaia', 'Cronus', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (4, 'Uranus', 'Cronus', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (5, 'Gaia', 'Rhea', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (6, 'Uranus', 'Rhea', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (7, 'Cronus', 'Zeus', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (8, 'Rhea', 'Zeus', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (9, 'Cronus', 'Hera', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (10, 'Rhea', 'Hera', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (11, 'Cronus', 'Demeter', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (12, 'Rhea', 'Demeter', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (13, 'Cronus', 'Poseidon', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (14, 'Rhea', 'Poseidon', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (15, 'Cronus', 'Hades', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (16, 'Rhea', 'Hades', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (17, 'Zeus', 'Athena', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (18, 'Zeus', 'Ares', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")

```

```

VALUES (19, 'Hera', 'Ares', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (20, 'Uranus', 'Aphrodite', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (21, 'Zeus', 'Hephaestus', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (22, 'Hera', 'Hephaestus', 'hasSon');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (23, 'Zeus', 'Persephone', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (24, 'Demeter', 'Persephone', 'hasDaughter');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (25, 'Zeus', 'Hera', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (26, 'Hera', 'Zeus', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (27, 'Hades', 'Persephone', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (28, 'Persephone', 'Hades', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (29, 'Aphrodite', 'Hephaestus', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (30, 'Hephaestus', 'Aphrodite', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (31, 'Cronus', 'Rhea', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (32, 'Rhea', 'Cronus', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (33, 'Uranus', 'Gaia', 'marriedTo');
INSERT INTO "GREEK_MYTHOLOGY"."RELATIONSHIPS"("KEY", "SOURCE", "TARGET", "TYPE")
VALUES (34, 'Gaia', 'Uranus', 'marriedTo');

```

9.3 Appendix C - Notation

The syntactic notation used in this guide is an extended version of BNF ("Backus Naur Form").

In a BNF language definition, each syntactic element of the language (known as a BNF nonterminal symbol) is defined by means of a production rule. This defines the element in terms of a formula consisting of the characters, character strings, and syntactic elements that can be used to form an instance of the formula.

The following table explains the symbols used and their meanings:

Symbol	Meaning
<>	A character string enclosed in angle brackets is the name of a syntactic element (BNF nonterminal) of the GraphScript 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. Curly braces indicate that the expression may be repeated zero or more times.

Symbol	Meaning
	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 contents of the innermost pair of these braces or brackets.
!!	Introduces normal English text. This is used when the definition of a syntactic element is not expressed in BNF.
-	The minus operator. The horizontal bar excludes the result of the formula following the bar from valid rules that are defined in front of the bar. The scope of this operator is equivalent to the scope of the alternative operator.

Spaces are used to separate syntactic elements. Multiple spaces and line breaks are treated as a single space. Apart from those symbols with special functions (see 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 these 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, which forms an instance of any syntactic element, can be generated from the BNF definition of that syntactic element by performing the following steps:

- 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.
- Replace each ellipsis, and the object it applies to, with one or more instances of that object.
- For every portion of the string enclosed in square brackets, either delete the brackets and their contents or change the brackets to braces.
- For every portion of the string enclosed in braces, apply steps 1 through 5 to the substring between the braces, then remove the braces.
- Apply steps 1 through 5 to any BNF nonterminal symbol that remains in the string.

The expansion or production is complete when no further nonterminal symbols remain in the character string.

The left normal form derivation of a character string CS in the source language character set from a BNF nonterminal NT is obtained by applying steps 1 through 5 above to NT, always selecting the leftmost BNF nonterminal in step 5.

Important Disclaimer for Features in SAP HANA



For information about the capabilities available for your license and installation scenario, refer to the [Feature Scope Description for SAP HANA](#).

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.

About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
 - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
 - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.

The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Gender-Related Language

We try not to use gender-specific word forms and formulations. As appropriate for context and readability, SAP may use masculine word forms to refer to all genders.

© 2019 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company. The information contained herein may be changed without prior notice.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

Please see <https://www.sap.com/about/legal/trademark.html> for additional trademark information and notices.