Narya

User Manual

Contents

1. Narya - Overview

Narya is a graph querying language designed by Ege Berkay Gülcan and Umut M. Köksaldı. This tutorial will familiarize the reader with the fundamentals of Narya. The tutorial will explain the features of Narya and its syntax.

2. Narya – Data Types

A data type is a data classification which tells a compiler or interpreter how the data is intended to be used which makes data types a necessary feature for all programming languages. Narya supports 3 built-in data types:

- integer: Specifies an integer.
- float: Specifies a floating point number which includes decimal numbers and integers.
- string: Specifies a sequence of characters. Can be empty as well. Indicated by a sequence of characters between two single quotation marks or two double quotation marks. If one needs to use a single or double quotation mark as a character, they need to put '\' before the character.

Narya has a dynamic type system; therefore the types of data are not specified explicitly.

3. Narya – Properties

A property is a name – value pair, separated by a comma (,) and enclosed within parentheses (()). A property name must always be a string type. However, the value of the property could be any data or collection type. Properties could be attached to vertices and edges. It is possible to attach multiple properties to a vertex or edge.

Example:

A property: ("ids", [1, 2, [3,4]])

4. Narya – Collection Types

Collection types, as the name suggests, are collections of data. The collected data could be any type or mix of types. There are 3 built-in collection types in Narya:

- List: Lists are comma separated values enclosed within square brackets ([]). Enclosed values could be any of the data or collection types. Lists accesses are done using integer indexes. The first item in the list is on the index 0. To access the value the user must use the name of the list. The name could be the property name if the list is a property value or the name of the collection type if it is enclosed within another collection type.

Examples:

Creating list as a property value:

ADD VERTEX v1 INTO graph_name WITH (“names”, **[“john”, “joe”]**);

Accessing first value of a list where it is a property value:

GET PATH (edge[1].**name[0] == “john”**) IN graph_name;

Accessing second value of a list where it is in another list:

GET PATH (edge[1].**values[0][1] == 2**) IN graph_name;

- Set: Sets are similar to lists, with the difference of not being able to use duplicate values in sets. Lists accesses and creation is also similar to lists. To define a list, the user must enter comma separated values that are enclosed within parentheses. For accesses the only difference is using parentheses (()) instead of square brackets ([]). Unlike lists, sets only accept set values and data types. Adding maps and lists is not supported.

  Examples:

  Creating set as a property value:

  ADD VERTEX v1 INTO graph_name WITH (“names”, **(“john”, “david”)**);

  Accessing first value of a set where it is a property value:

  GET PATH (edge[3].**name(0) == “kevin”**) IN graph_name;

  Accessing second value of a set where it is in a list:

  GET PATH (edge[7].**values[0](1) == 1.84**) IN graph_name;

- Map: Maps are collected key, value pairs. The values could be any data or collection types; however keys could only be data types or sets. To create a map, the programmer must enter key value pairs, separated by a colon (:), that are enclosed within curly brackets ({}). Access is similar to lists, however instead of integer indexes, the key should be placed within the square brackets ([]).

  Examples:

  Creating map as a property value:

  ADD VERTEX v1 INTO graph_name WITH (“values”, **{“name” : “john”, “age” : 21}**);

  Accessing a value of a map where it is a property value:

  GET PATH (edge[2].**values[“name”] == “david”**) IN graph_name;

  Accessing a value of a map where it is in a list:

  GET PATH (edge[5].**values[0][“age”] == 35**) IN graph_name;

5. Narya – Variables

Narya supports using variables in statements. Variable names should start with a character or an underscore (_) and should be continued by characters, digits and underscores. Since Narya uses a dynamic type system there is no need to specify the type name.

Examples:

_ab2 = 3;

ERT = {“key” : “value”};

aB12_ = 17.8;

6. Narya – Operators

Narya operators are reserved characters that are mainly used in path queries to perform operations such as arithmetic and logical operations. There are 4 types of operators in Narya:

- Arithmetic operators
- Comparison operators
- Logical operators
- Query operators

6.1. Arithmetic Operators

Operator	Description	Example
+(Addition)	Adds values on either side of the operator	10 + 20 will give 30
-(Subtraction)	Subtracts right hand operand from the left hand operand	55 – 48 will give 7
*(Multiplication)	Multiplies values on either side of the operator	8 * 14 will give 112
/(Division)	Divides left hand operand by right hand operand	38 / 2 will give 19
%(Modulus)	Divides left hand operand by right hand operand and returns remainder	13 % 2 will give 1

6.2. Comparison Operators

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(13 == 13) is true
!=	Checks if the values of two operands are equal or not, if not then condition becomes true.	(13 != 13) is false
<	Checks if the value of the left operand is smaller than the right operand, if yes then condition becomes true.	(32 < 53) is true
>	Checks if the value of the left operand is greater than the right operand, if yes then condition becomes true.	(82 > 17) is true
<=	Checks if the value of the left operand is smaller than or equal to the right operand, if yes then condition becomes true.	(23 <= 23) is true
>=	Checks if the value of the left operand is greater than or equal to the right operand, if yes then condition becomes true.	(2 >= 41) is false

6.3. Logical Operators

Operator	Description	Example
&&	The operator returns true if both conditions hold true	(13 == 13) && (2 < 11) is true
||	The operator returns true if either one of the conditions hold true	(4 > 28) || (6 >= 6) is true
!	The operator negates the result of the condition	!(7 < 11) is false

6.4.Query Operators

Operator	Description	Example
&(Concatenation)	Concatenates the result of the right path query to the result of the left	GET PATH((p_length == 3)&(edge(1).name == “index”)) IN graph_name;
|(Alternation)	Returns the paths generated from either of the path queries	GET PATH((p_length == 7) | (edge(1).name == “abc”)) IN graph_name;
#(Repetition)	Returns the paths that satisfy the path query zero or more times(similar to regular expressions)	GET PATH((edge(‘abc’).name == “asdf”) IN graph_name;

7. Narya – Define Graph

Narya has 2 graph types, which are directed and undirected graphs. To create a new graph **CREATE** keyword is used. To determine the graph type the keywords **DIRECTED** and **UNDIRECTED** are used. All of the keywords of Narya are case insensitive, meaning that both **CREATE** and **create** are accepted. In addition, all the statements must end with a semicolon (;).

Examples:

CREATE UNDIRECTED graph_name;

CREATE DIRECTED graph_name2;

8. Narya – Add Vertex

To add a new vertex, **ADD VERTEX** keyword is used. After **ADD VERTEX** the programmer must specify a name for the vertex. Vertex names consist of characters, digits and underscores (_).The vertex names are case sensitive. After the vertex name by using the **INTO** keyword and a graph name the graph which the vertex will be added should be stated. Lastly, to add vertex properties, after the graph name the keyword **WITH** and the comma separated property list which is enclosed within parentheses.

Example:

ADD VERTEX v1 INTO graph_name WITH ((“name”, “first”),(“id”, {“index” : 1}));

9. Narya – Add Edge

To add a new edge, **ADD EDGE** keyword is used. Following the **ADD EDGE** keyword the user must specify an edge name which consists of digits, characters and underscores (_). After the edge name, **FROM** vertex1_name **TO** vertex2_name, should be used to specify the beginning and ending vertices of the edge. In a directed graph the direction is from beginning to the ending vertex. Following the second vertex name, similar to vertex addition, graph name and properties should be stated using **INTO** and **WITH** keywords.

Example:

```
ADD EDGE FROM v1 TO v2 INTO graph_name WITH ((“id”, 1), (“names”, [“john”, “doe”]), (“index”, 3));
```

10. Narya – Query Graph

To query a graph the keyword **GET PATH** should be used. After that a path query should be entered within parentheses (()) and after the closing parenthesis the keyword **IN** and the graph name should be added to specify which graph to search in.

A path query is comma separated conditions which will be used when searching for a path. There are 4 special keywords to use while constructing path queries. **EDGE[edge_name]** is used to access the edge with name **edge_name**. Instead of using the name, an index could be used to access the edges as well. In this case, the first (0th) edge will be the first edge added to the graph. The indexes could be ranges as well such as **EDGE[1-4]** to access a range of edges. In order to access all of the edges the character **‘#’** could be used instead of **edge_name**. The properties of the edge could be accessed by placing a dot (.) after getting edge and using the property name. In order to access all of the properties of the edge the character **‘#’** could be used instead of the property name. If the desired property does not exist, the query returns nothing.

Each edge has one beginning and one ending vertex. These vertices could be accessed by first obtaining the edge by **EDGE[edge_name]**, placing a dot after it and using the keywords **V_B** for beginning vertex and **V_E** for ending vertex. The vertex properties could be accessed by placing a dot (.) after getting the vertex and using the property name. In order to access all of the properties of the vertex the character **‘#’** could be used instead of the property name. If the desired property does not exist, the query returns nothing.

Lastly, the keyword **P_LENGTH** could be used to refer the path length of the generated path.

Examples:

Get paths with length 3, 2nd edge property name = “joe” or 2nd edge property name = “david” from graph g1:

```
GET PATH(p_length == 3, (edge[1].name == “joe” || EDGE[1].name == “david”)) IN g1;
```

Get all paths which has index == 1 for each beginning vertex for every 3rd edge from graph g2:

```
GET PATH((P_LENGTH == 3, EDGE[0].v_b.index == 1)#) IN g2;
```

11. Narya – Built-in Functions

Narya has 6 built-in functions that allow the programmer to create more complicated queries. These could be used anywhere in a path query. The parameters are entered as comma separated values within parentheses after the function name.

11.1. SUM(property_name, integer)

SUM function takes a property name and an integer as parameters. The integer determines where to look for that parameter. If it is 0, the vertices should be searched and if it is 1, edges are searched. The functions sums the value of the properties with the given name in the given graph element and returns the result. If there are unmatching types, such as string and integer, returns -1.

11.2. MIN(property_name, integer)

MIN function takes a property name and an integer as parameters. The integer determines where to look for that parameter. If it is 0, the vertices should be searched and if it is 1, edges are searched. The return value is the minimum value of the given property. If there are unmatching types returns -1.

11.3. MAX(property_name, integer)

MAX function takes a property name and an integer as parameters. The integer determines where to look for that parameter. If it is 0, the vertices should be searched and if it is 1, edges are searched. The return value is the maximum value of the given property. If there are unmatching types returns -1.

11.4. AVG(property_name, integer)

AVG function takes a property name and an integer as parameters. The integer determines where to look for that parameter. If it is 0, the vertices should be searched and if it is 1, edges are searched. The return value is the average value of the given property. If there are unmatching types returns -1.

11.5. EXISTS(property_name, element)

EXISTS function takes a property name and a graph element as parameters. If that property exists within that element, it returns true and false otherwise.

11.6. SUBSTR(string, start, end)

SUBSTR function takes a string and 2 integers as parameters. The integers represent the start and end points of the substring. The return value is a new string starting at the **start** index of the **string** string and ending at **end** index.