NebulaGraph

Nebula Graph Database Manual

v1.2.1

Min Wu, Amber Zhang, XiaoDan Huang

Table of contents

1. Overview

1.1 About This Manual

This is the **Nebula Graph** User Manual. It documents **Nebula Graph** 1.2.1. For information about which versions have been released, see Release Notes.

1.1.1 Who Shall Read This Manual

This manual is written for `algorithms engineers`, `data scientists`, `software developers`, `database administrators`, and all the people who are interested in the `Graph Database` areas.

If you have questions about using **Nebula Graph**, join the Nebula Graph Community Slack or our Official Forum.

If you have suggestions concerning additions or corrections to the manual itself, please do not hesitate to open an issue on GitHub.

1.1.2 Syntax Conventions

Nebula Graph is under constant development, and this User Manual is updated frequently as well.

This manual uses certain typographical conventions:

- `Fixed width`

  A fixed-width font is used for `ngql statements`, `code examples`, `system output`, and `file names`.

- **Bold**

  Bold typeface indicates **commands**, **user types**, or **interface**.

- `UPPERCASE`

  `REVERSED KEYWORDS` and `NON REVERSED KEYWORDS` in query-language and code examples are almost always shown in upper case.

1.1.3 File Formats

The manual source files are written in Markdown format.

The HTML version is produced by mkdocs.

Last update: April 15, 2021

1.2 Welcome to Nebula Graph 1.2.1 Documentation

Nebula Graph is a distributed, scalable, and lightning-fast graph database.

It is the optimal solution in the world capable of hosting graphs with dozens of billions of vertices (nodes) and trillions of edges with millisecond latency.

1.2.1 Tutorial Video

- YouTube
- bilibili

1.2.2 Preface

- About This Manual
- Manual Change Log

1.2.3 Overview (for Beginners)

- Introduction
- Concepts
  - Data Model
  - Query Language Overview
- Quick Start and Useful Links
  - Get Started
  - FAQ
  - Build Source Code
  - Import .csv File
  - Nebula Graph Clients
- Design and Architecture
  - Design Overview
  - Storage Architecture
  - Query Engine

1.2.4 Query Language (for All Users)

- Data Types
  - Data Types
  - Type Conversion

- Functions and Operators
  - Bitwise Operators
  - Built-In Functions
  - Comparison Functions And Operators
  - Group By Function
  - Limit Syntax
  - Logical Operators
  - Order By Function
  - Set Operations
  - String Comparison Functions and Operators
  - uuid Function
- Language Structure
  - Literal Values
    - Boolean Literals
    - Numeric Literals
    - String Literals
  - Comment Syntax
  - Identifier Case Sensitivity
  - Keywords and Reserved Words
  - Pipe Syntax
  - Property Reference
  - Schema Object Names
  - Statement Composition
  - User-Defined Variables

- Statement Syntax
  - Data Definition Statements
    - Alter Edge Syntax
    - Alter Tag Syntax
    - Create Space Syntax
    - Create Edge Syntax
    - Create Tag Syntax
    - Drop Edge Syntax
    - Drop Space Syntax
    - Drop Tag Syntax
    - Index
    - TTL (time-to-live)
  - Data Query and Manipulation Statements
    - Delete Edge Syntax
    - Delete Vertex Syntax
    - Fetch Syntax
    - Go Syntax
    - Insert Edge Syntax
    - Insert Vertex Syntax
    - Lookup Syntax
    - Return Syntax
    - Update Edge Syntax
    - Update Vertex Syntax
    - Upsert Syntax
    - Where Syntax
    - Yield Syntax
  - Utility Statements
    - Show Statements
      - Show Charset Syntax
      - Show Collation Syntax
      - Show Configs Syntax
      - Show Create Spaces Syntax
      - Show Create Tag/Edge Syntax
      - Show Hosts Syntax
      - Show Indexes Syntax
      - Show Parts Syntax
      - Show Roles Syntax
      - Show Snapshots Syntax
      - Show Spaces Syntax
      - Show Tag/Edge Syntax
      - Show Users Syntax
    - Describe Syntax
    - Use Syntax

1.2.5 Build Develop and Administration (for Developers and DBA)

- Monitoring and Statistics
  - Metrics
  - Meta Metrics
  - Storage Metrics
  - Graph Metrics
  - RocksDB Statistics
  - Development and API
  - Nebula Graph Clients

1.2.6 Contributions (for Contributors)

- Contribute to Documentation
- Cpp Coding Style
- How to Contribute

1.2.7 Appendix

- Gremlin V.S. nGQL
- Cypher V.S. nGQL
- SQL V.S. nGQL
- Vertex Identifier and Partition

Last update: April 15, 2021

1.3 Concepts

1.3.1 Graph Data Modeling

This guide is designed to walk you through the graph data modeling of **Nebula Graph**. Basic concepts of designing a graph data model will be introduced.

Graph Space

Graph Space is a physically isolated space for different graph. It is similar to database in MySQL.

Directed Property Graph

Nebula Graph stores data in directed property graphs. A directed property graph is a graph of vertices and edges that have a direction. It can be described as follows:

G = < V, E, P_V, P_E >

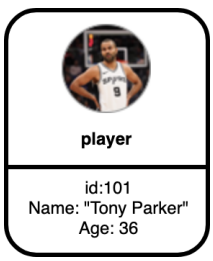
- **V** is the collection of vertices.
- **E** is the collection of edges.
- P_V represents vertex properties.
- P_E represents edge properties.

The following table describes a dataset `basketballplayer`. It has two types of vertices, i.e., **player** and **team**, and two types of edges, i.e., **serve** and **follow**.

Type	Name	Property (data type)	Description
tag	player	name string age int	Represents players.
tag	team	name string	Represents teams.
edge type	serve	start_year int end_year int	Represents the behavior of players that connects players with teams. The direction of the serve edges is from players to teams.
edge type	follow	degree int	Represents the behavior of players that connects players. The direction of the follow edges is from one player to another player.

Vertices

Vertices are typically used to represent entities in the real world. In **Nebula Graph**, vertices are identified with vertex identifiers (i.e. VIDs). The `VID` must be unique in the graph space. In the preceding example, the graph contains eleven vertices.

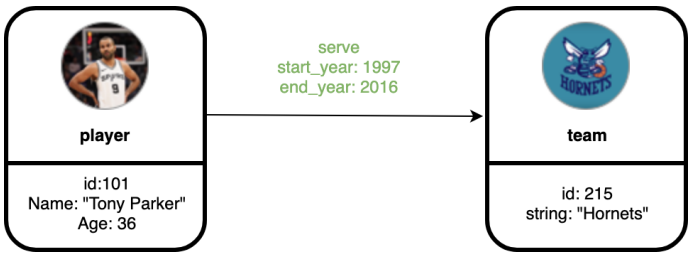


Tags

In **Nebula Graph**, vertex properties are clustered by **tags**. One vertex can have one or more tags. In the preceding example, the vertices have tags **player** and **team**.

Edge

Edges are used to connect vertices. Each edge usually represents a relationship or a behavior between two vertices. In the preceding example, edges are ***serve*** and ***follow***.



Edge Type

`Each edge` is an instance of an edge type. Our example uses ***serve*** and ***follow*** as edge types. Take edge ***serve*** for example, in the preceding picture, vertex `101` (represents a **player**) is the source vertex and vertex `215` (represents a **team**) is the target vertex. We see that vertex `101` has an outgoing edge while vertex `215` has an incoming edge.

Properties of Vertices and Edges

Both vertices and edges can have properties. Properties are described with key value pairs. In our example graph, we have used the properties `id`, `name` and `age` on **player**, `id` and `name` on **team**, and `degree` on **follow** edge.

Edge Rank

Edge rank is an immutable user-assigned 64-bit signed integer. It affects the edge order of the same edge type between two vertices. The edge with a higher rank value comes first. When not specified, the default rank value is zero. The current sorting basis is "binary coding order", i.e. 0, 1, 2, ... 9223372036854775807, -9223372036854775808, -9223372036854775807, ..., -1. In addition to an edge type, the edge between two vertices must have an edge rank. The edge rank is a 64-bit integer assigned by the user; if not specified, the edge rank defaults to 0.

An edge can be represented uniquely with the [source vertex, edge type, edge rank, destination vertex].

The edge rank affects the edge order of the same edge type between two vertices. The edge with a higher rank value comes first.

The current sorting basis is "binary coding order", i.e. 0, 1, 2, ... 9223372036854775807, -9223372036854775808, -9223372036854775807, ..., -1.

Schema

In **Nebula Graph**, schema refers to the definition of properties (name, type, etc.). Like `MySQL`, **Nebula Graph** is a **strong typed** database. The name and data type of the properties should be determined before the data is written.

Last update: April 16, 2021

1.3.2 Nebula Graph Query Language (nGQL)

About nGQL

`nGQL` is a declarative, textual query language like SQL, but for graphs. Unlike SQL, nGQL is all about expressing graph patterns. nGQL is a work in progress. We will add more features and further simplify the existing ones. There might be inconsistency between the syntax specs and implementation for the time being.

Goals

- Easy to learn
- Easy to understand
- To focus on the online queries, also to provide the foundation for the offline computation

Features

- Syntax is close to SQL, but not exactly the same (Easy to learn)
- Expandable
- Case insensitive
- Support basic graph traverse
- Support pattern match
- Support aggregation
- Support graph mutation
- Support distributed transaction (future release)
- Statement composition, but **NO** statement embedding (Easy to read)

Terminology

- **Graph Space** : A physically isolated space for different graph
- **Tag** : A label associated with a list of properties
  - Each tag has a name (human readable string), and internally each tag will be assigned a 32-bit integer
  - Each tag associates with a list of properties, each property has a name and a type
  - There could be dependencies between tags. The dependency is a constrain, for instance, if tag S depends on tag T, then tag S cannot exist unless tag T exists
- **Vertex** : A Node in the graph
  - Each vertex has a unique 64-bit (signed integer) ID (**VID**)
  - Each vertex can associate with multiple **tags**
- **Edge** : A Link between two vertices
  - Each edge can be uniquely identified by a tuple
  - ***Edge type (ET)*** is a human readable string, internally it will be assigned a 32-bit integer. The edge type decides the property list (schema) on the edge.
  - ***Edge rank*** is an immutable user-assigned 64-bit signed integer. It affects the edge order of the same edge type between two vertices. The edge with a higher rank value comes first. When not specified, the default rank value is zero. The current sorting basis is "binary coding order", i.e. 0, 1, 2, ... 9223372036854775807, -9223372036854775808, -9223372036854775807, ..., -1.
  - Each edge can only be of one type

- **Path** : A *non-forked* connection with multiple vertices and edges between them
  - The length of a path is the number of the edges on the path, which is one less than the number of vertices
  - A path can be represented by a list of vertices, edge types, and rank. An edge is a special path with length==1

```
<vid, <edge_type, rank>, vid, ...>
```

Language Specification at a Glance

For most readers, You can skip this section if you are not familiar with BNF.

GENERAL

- The entire set of statements can be categorized into three classes: **query**, **mutation**, and **administration**
- Every statement can yield a data set as the result. Each data set contains a schema (column name and type) and multiple data rows

COMPOSITION

- Statements could be composed in two ways:
  - Statements could be piped together using operator "**|**", much like the pipe in the shell scripts. The result yielded from the previous statement could be redirected to the next statement as input
  - More than one statements can be batched together, separated by "**;**". The result of the last statement (or a **RETURN** statement is executed) will be returned as the result of the batch

DATA TYPES

- Simple type: **vid**, **double**, **int**, **bool**, **string**, **timestamp**
- **vid**: 64-bit signed integer, representing a vertex ID

TYPE CONVERSION

- A simple typed value can be implicitly converted into a list
- A list can be implicitly converted into a one-column tuple list
  - "<type>_list" can be used as the column name

COMMON BNF

::= **vid** | **integer** | **double** | **float** | **bool** | **string** | **path** | **timestamp** | **year** | **month** | **date** | **datetime**

::=

<type> ::= |

::= **vid** (, **vid**)* | "{" **vid** (, **vid**)* "}"

<label> ::= [:alpha] ([:alnum:] | "_")*

::= ("_")* <label>

::= <label>

::= (,)*

::= :<type>

::= ":"

::=

::= <tuple> (, <tuple>)* | "{" <tuple> (, <tuple>)* "}"

<tuple> ::= "(" **VALUE** (, **VALUE**)* ")"

<var> ::= "$" <label>

STATEMENTS

Choose a Graph Space

Nebula supports multiple graph spaces. Data in different graph spaces are physically isolated. Before executing a query, a graph space needs to be selected using the following statement

USE

Return a Data Set

Simply return a single value or a data set

RETURN

::= **vid** | | | <var>

Create a Tag

The following statement defines a **new** tag

CREATE TAG ()

::= <label>
::= +
::= ,<type>
::= <label>

Create an Edge Type

The following statement defines a **new** edge type

CREATE EDGE ()

:= <label>

Insert Vertices

The following statement inserts one or more vertices

INSERT VERTEX [**NO OVERWRITE**] **VALUES**

::= () (, ())*
::= :() (, :())*
::= **vid**
::= (,)*
::= **VALUE** (, **VALUE**)*

Insert Edges

The following statement inserts one or more edges

INSERT EDGE [**NO OVERWRITE**] [()] **VALUES** ()+

edge_value ::= -> [@ <rank>] :

Update a Vertex

The following statement updates a vertex

UPDATE VERTEX SET \<update_decl> [**WHERE** <conditions>] [**YIELD**]

::= |
::= = <expression> {, = <expression>}+
::= () = () | () = <var>

Update an Edge

The following statement updates an edge

UPDATE EDGE -> [@<rank>] **OF SET** [**WHERE** <conditions>] [**YIELD**]

Traverse the Graph

Navigate from given vertices to their neighbors according to the given conditions. It returns either a list of vertex IDs, or a list of tuples

GO [**STEPS**] **FROM** [**OVER** [**REVERSELY**]] [**WHERE**] [**YIELD**]

::= [data_set] [[**AS**] <label>]
::= **vid** | | | <var>
::= [**AS** <label>] ::= {, }*
::= <label>

::= <filter> {**AND** | **OR** <filter>}*
::= \ **>**\ | **>= | < | <= | == | != <expression> | <expression> IN <value_list>
::= {, }*
::= <expression> [AS** <label>]

WHERE clause only applies to the results that are going to be returned. It will not be applied to the intermediate results (See the detail description of the **STEP[S]** clause)

When **STEP[S]** clause is skipped, it implies **one step**

When going out for one step from the given vertex, all neighbors will be checked against the **WHERE** clause, only results satisfied the **WHERE** clause will be returned

When going out for more than one step, **WHERE** clause will only be applied to the final results. It will not be applied to the intermediate results. Here is an example

```
GO 2 STEPS FROM me OVER friend WHERE birthday > "1988/1/1"
```

Obviously, you will probably guess the meaning of the query is to get all my fof (friend of friend) whose birthday is after 1988/1/1. You are absolutely right. We will not apply the filter to my friends (in the first step).

Search

Following statements looks for vertices or edges that match certain conditions

FIND VERTEX WHERE [**YIELD**]

FIND EDGE WHERE [**YIELD**]

PROPERTY REFERENCE

It's common to refer a property in the statement, such as in **WHERE** clause and **YIELD** clause. In nGQL, the reference to a property is defined as

::= <object> "."
<object> ::= | | <var>
::= <label>
::= '[' ']'

<var> always starts with "$". There are two special variables: $- and $$.

$- refers to the input stream, while $$ refers to the destination objects

All property names start with a letter. There are a few system property names starting with "_". All properties names starting with "_" are reserved.

BUILT-IN PROPERTIES

- _id : Vertex id
- _type : Edge type
- _src : Source ID of the edge
- _dst : Destination ID of the edge
- _rank : Edge rank number

Last update: April 8, 2021

1.4 Quick Start

1.4.1 Quick Start

This guide walks you through the process of using **Nebula Graph**:

- Install
- Data Schema
- CRUD
- Batch inserting
- Data Import Tools

Installing Nebula Graph

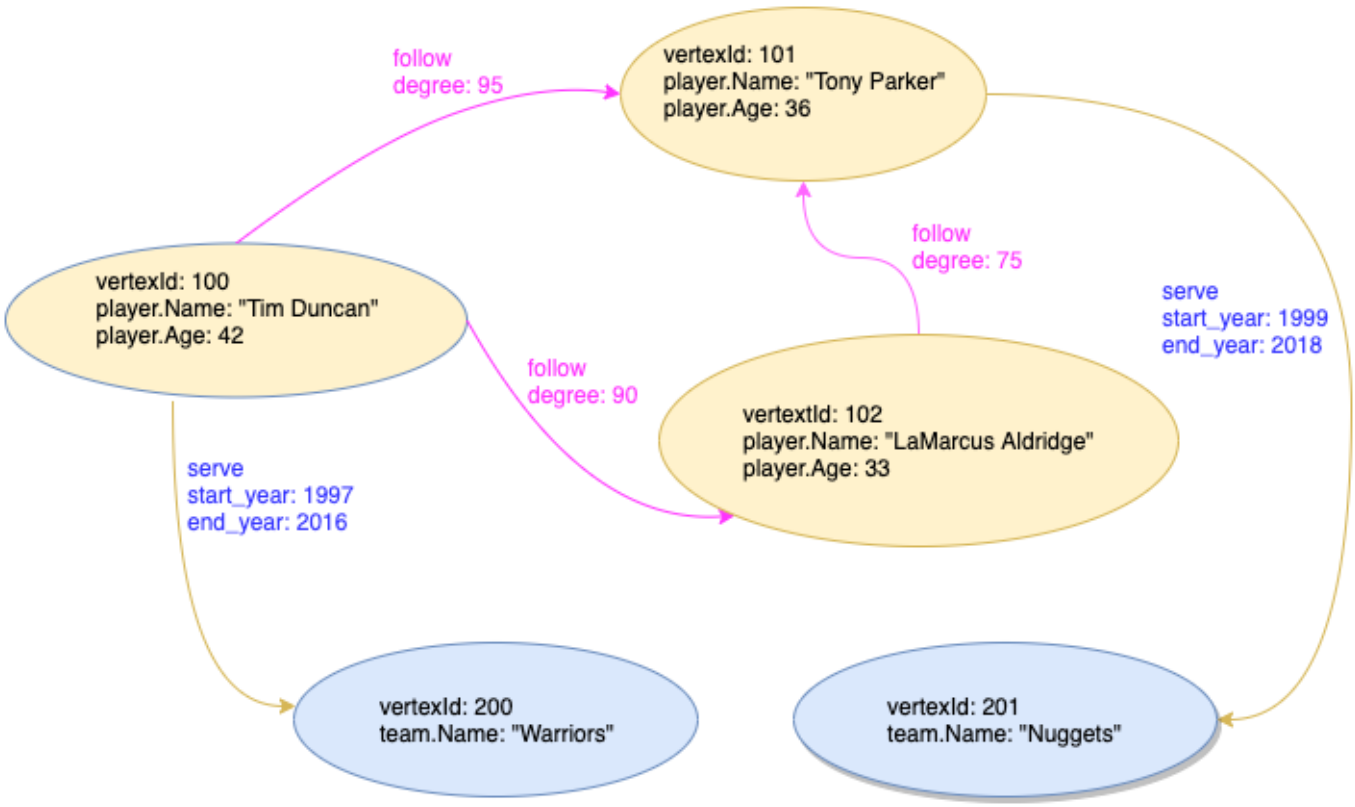
It is recommended to install Nebula Graph with Docker Compose. We have recorded a quick video tutorial on YouTube for your reference.

In addition to Docker, you can also install **Nebula Graph** by installing source code or rpm/deb packages.

In case you encounter any problem during the installation, be sure to ask us on our official forum. We have developers on call to answer your questions.

Data Schema

In this guide, all operations are based on the `basketballplayer` dataset below:



In the above figure, there are two tags (**player**, **team**) and two edge types (**serve** and **follow**).

CREATING AND USING A GRAPH SPACE

A graph space in **Nebula Graph** is similar to an individual database that you create in traditional databases such as MySQL. First, you need to create a space and use it before can do any other operations.

You can create and use a graph space by the following steps:

1. Check the cluster machine status:

```
nebula> SHOW HOSTS;
=============================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=============================================================================================
| 192.168.8.210 | 44500 | online |              |                     |                        |
---------------------------------------------------------------------------------------------
| 192.168.8.211 | 44500 | online |              |                     |                        |
---------------------------------------------------------------------------------------------
```

   The status `online` indicates that the **storage service** `storaged` has successfully connected to the **metadata service process** `metad`.

2. Enter the following statement to create a graph space:

```
nebula> CREATE SPACE basketballplayer(partition_num=10, replica_factor=1);
```

   Here:

   - `partition_num` specifies the number of partitions in one replica. It is usually 5 times the number of hard disks in the cluster.
   - `replica_factor` specifies the number of replicas in the cluster. It is usually 3 in production, 1 in test. Due to the majority voting principle, it must set to be odd.

   You can also check machine and partition distribution with `SHOW HOSTS` statement:

   ```ngql
   nebula> SHOW HOSTS;
   ```

   ```
   ==========================================================================================
   | Ip | Port | Status | Leader count | Leader distribution | Partition distribution |
   ==========================================================================================
   | 192.168.8.210 | 44500 | online | 8 | basketballplayer: 8 | test: 8 |
   ------------------------------------------------------------------------------------------
   | 192.168.8.211 | 44500 | online | 2 | basketballplayer: 2 | test: 2 |
   ------------------------------------------------------------------------------------------
   ```

   ```
   If all the machines are online, but the leader distribution is unbalanced (see above), re-distribute partitions with `BALANCE LEADER` statement:

   ```ngql
   nebula> BALANCE LEADER;
   =============================================================================================================
   | Ip            | Port  | Status | Leader count | Leader distribution             | Partition distribution |
   =============================================================================================================
   | 192.168.8.210 | 44500 | online | 5            | basketballplayer: 5             | test: 5                |
   -------------------------------------------------------------------------------------------------------------
   | 192.168.8.211 | 44500 | online | 5            | basketballplayer: 5             | test: 5                |
   -------------------------------------------------------------------------------------------------------------
   ```

   See here for details.

3. Enter the following statement to use the graph space:

```
nebula> USE basketballplayer;
```

4. Now you can check the graph space you just created:

```
nebula> SHOW SPACES;
```

   The following information is returned:

```
====================
| Name             |
====================
| basketballplayer  |
--------------------
```
```

DEFINING THE SCHEMA FOR YOUR DATA

In **Nebula Graph**, we classify different vertices with similar properties into one group which is named a tag. The `CREATE TAG` statement defines a tag with a tag name followed by properties and the property types enclosed in parentheses. The `CREATE EDGE` statement defines an edge type with a type name, followed by properties and the property types enclosed in parentheses.

You can create tags and edge types by the following steps:

1. Enter the following statement to create the **player** tag:

```
nebula> CREATE TAG player(name string, age int);
```

2. Enter the following statement to create the **team** tag:

```
nebula> CREATE TAG team(name string);
```

3. Enter the following statement to create the **follow** edge type:

```
nebula> CREATE EDGE follow(degree int);
```

4. Enter the following statement to create the **serve** edge type:

```
nebula> CREATE EDGE serve(start_year int, end_year int);
```

5. Now you can check the tags and edge types you just created.

   5.1 To show the tags you just created, enter the following statement:

```
nebula> SHOW TAGS;
```

   The following information is returned:

```
============
| Name     |
============
| player   |
------------
| team     |
------------
```

   5.2 To show the edge types you just created, enter the following statement:

```
nebula> SHOW EDGES;
```

   The following information is returned:

```
==========
| Name   |
==========
| serve  |
----------
| follow |
----------
```

   5.3 To show the properties of the **player** tag, enter the following statement:

```
nebula> DESCRIBE TAG player;
```

   The following information is returned:

```
===================
| Field  | Type   |
===================
| name   | string |
-------------------
| age    | int    |
-------------------
```

   5.4 To show the properties of the **follow** edge type, enter the following statement:

```
nebula> DESCRIBE EDGE follow;
```

   The following information is returned:

```
=====================
| Field    | Type   |
=====================
| degree   | int    |
---------------------
```

CRUD

INSERTING DATA

You can insert vertices and edges based on relations in the illustration figure.

Inserting Vertices

The `INSERT VERTEX` statement inserts a vertex by specifying the vertex tag, properties, vertex ID and property values.

You can insert some vertices by the following statements:

```
nebula> INSERT VERTEX player(name, age) VALUES 100:("Tim Duncan", 42);
nebula> INSERT VERTEX player(name, age) VALUES 101:("Tony Parker", 36);
nebula> INSERT VERTEX player(name, age) VALUES 102:("LaMarcus Aldridge", 33);
nebula> INSERT VERTEX team(name) VALUES 200:("Warriors");
nebula> INSERT VERTEX team(name) VALUES 201:("Nuggets");
nebula> INSERT VERTEX player(name, age) VALUES 121:("Useless", 60);
```

- In the above vertices inserted, the number after the keyword `VALUES` is the vertex ID (abbreviated for `VID`, int64). The `VID` must be unique in the space.
- The last vertex (VID: 121)inserted will be deleted in the deleting data section.
- If you want to insert multiple vertices for the same tag by a single `INSERT VERTEX` operation, you can enter the following statement:

```
nebula> INSERT VERTEX player(name, age) VALUES 100:("Tim Duncan", 42), \
101:("Tony Parker", 36), 102:("LaMarcus Aldridge", 33);
```

Inserting Edges

The `INSERT EDGE` statement inserts an edge by specifying the edge type name, properties, source vertex VID and target vertex VID, and property values.

You can insert some edges by the following statements:

```
nebula> INSERT EDGE follow(degree) VALUES 100 -> 101:(95);
nebula> INSERT EDGE follow(degree) VALUES 100 -> 102:(90);
nebula> INSERT EDGE follow(degree) VALUES 102 -> 101:(75);
nebula> INSERT EDGE serve(start_year, end_year) VALUES 100 -> 200:(1997, 2016);
nebula> INSERT EDGE serve(start_year, end_year) VALUES 101 -> 201:(1999, 2018);
```

Similarly: If you want to insert multiple edges for the same edge type by a single `INSERT EDGES` operation, you can enter the following statement:

```
nebula> INSERT EDGE follow(degree) VALUES 100 -> 101:(95),100 -> 102:(90),102 -> 101:(75);
```

FETCHING DATA

After you insert some data in **Nebula Graph**, you can retrieve any data from your graph space.

The `FETCH PROP ON` statement retrieve data from your graph space. If you want to fetch vertex data, you must specify the vertex tag and vertex VID; if you want to fetch edge data, you must specify the edge type name, source vertex VID and target vertex VID.

To fetch the data of the player whose `VID` is `100`, enter the following statement:

```
nebula> FETCH PROP ON player 100;
```

The following information is returned:

```
=====================================
| VertexID | player.name | player.age |
=====================================
| 100      | Tim Duncan  | 42         |
-------------------------------------
```

To fetch the data of the **serve** edge between `VID` `100` and `VID` `200`, enter the following statement:

```
nebula> FETCH PROP ON serve 100 -> 200;
```

The following information is returned:

```
==========================================================================
| serve._src | serve._dst | serve._rank | serve.start_year | serve.end_year |
==========================================================================
| 100        | 200        | 0           | 1997             | 2016           |
--------------------------------------------------------------------------
```

UPDATING DATA

You can update the vertices and edges you just inserted.

Updating Vertices

The `UPDATE VERTEX` statement updates data for your vertex by selecting the vertex that you want to update and then setting the property value with an equal sign to assign it a new value.

The following example shows you how to change the `name` value of `VID 100` from `Tim Duncan` to `Tim`.

Enter the following statement to update the `name` value:

```
nebula> UPDATE VERTEX 100 SET player.name = "Tim";
```

To check whether the `name` value is updated, enter the following statement:

```
nebula> FETCH PROP ON player 100;
```

The following information is displayed:

```
=====================================
| VertexID | player.name | player.age |
=====================================
| 100      | Tim         | 42         |
-------------------------------------
```

Updating Edges

The `UPDATE EDGE` statement updates data for your edge by specifying the source vertex ID and the target vertex ID of the edge and then setting the property value with an equal sign to assign it a new value.

The following example shows you how to change the value of the `degree` property in the **follow** edge between `VID 100` and `VID 101`. Now we change the `degree` property from `95` to `96`.

Now we add the `degree` value by 1:

```
nebula> UPDATE EDGE 100 -> 101 OF follow SET degree = 96;
```

To check whether the `degree` value is updated, enter the following statement:

```
nebula> FETCH PROP ON follow 100 -> 101;
```

The following information is returned:

```
============================================================
| follow._src | follow._dst | follow._rank | follow.degree |
============================================================
| 100         | 101         | 0            | 96            |
------------------------------------------------------------
```

UPSERT

`UPSERT` is used to insert a new vertex or edge or update an existing one. If the vertex or edge doesn't exist it will be created. `UPSERT` is a combination of `INSERT` and `UPDATE`.

Consider the following example:

```
nebula> INSERT VERTEX player(name, age) VALUES 111:("Ben Simmons", 22); -- Insert a new vertex.
nebula> UPSERT VERTEX 111 SET player.name = "Dwight Howard", player.age = $^.player.age + 11 WHEN $^.player.name == "Ben Simmons" && $^.player.age > 20 YIELD
```

```
$^.player.name AS Name, $^.player.age AS Age; -- Do upsert on the vertex.
======================
| Name          | Age |
======================
| Dwight Howard | 33  |
----------------------
```

See UPSERT Doc for details.

DELETING DATA

If you have some data that you do not need, you can delete it from your graph space.

Deleting Vertices

You can delete any vertex from your graph space. The `DELETE VERTEX` statement deletes a vertex by specifying the vertex VID.

To delete a vertex whose `VID` is `121`, enter the following statement:

```
nebula> DELETE VERTEX 121;
```

To check whether the vertex is deleted, enter the following statement;

```
nebula> FETCH PROP ON player 121;
```

The following information is returned:

```
Execution succeeded (Time spent: 1571/1910 us)
```

The above information with an empty return result indicates the query operation is successful but no data is queried from your graph space because the data is deleted.

Deleting Edges

You can delete any edge from your graph space. The `DELETE EDGE` statement deletes an edge by specifying the edge type name and the source vertex ID and target vertex ID.

To delete a **follow** edge between `VID 100` and `VID 200`, enter the following statement:

```
nebula> DELETE EDGE follow 100 -> 200;
```

NOTE: If you delete a vertex, all the out-going and in-coming edges of this vertex are deleted.

Creating Index

You can add indexes to existing tag/edge-type with the `CREATE INDEX` keyword.

```
nebula> CREATE TAG INDEX player_index_0 on player(name);
```

The above statement creates an index for the *name* property on all vertices carrying the *player* tag.

```
nebula> REBUILD TAG INDEX player_index_0 OFFLINE;
```

Creating index will affect the write performance. We suggest you import data first and then rebuild the index in batch.

See the Index Documentation for details.

Sample Queries

This section gives more query examples for your reference.

EXAMPLE 1

Find the vertices that `VID 100` follows.

Enter the following statement:

```
nebula> GO FROM 100 OVER follow;
```

The following information is returned:

```
==============
| follow._dst |
==============
| 101         |
--------------
| 102         |
--------------
```

EXAMPLE 2

Find the vertex that `VID 100` follows, whose age is greater than `35`. Return his name and age, and set the column names to **Teammate** and **Age** respectively.

Enter the following statement:

```
nebula> GO FROM 100 OVER follow WHERE $$.player.age >= 35 \
YIELD $$.player.name AS Teammate, $$.player.age AS Age;
```

The following information is returned:

```
====================
| Teammate    | Age |
====================
| Tony Parker | 36  |
--------------------
```

Here:

- `YIELD` specifies what values or results you want to return from the query.
- `$$` represents the target vertex.
- `\` represents a line break.

EXAMPLE 3

Find the team which is served by the player who is followed by `100`.

There are two ways to get the same result. First, we can use a `pipe` to retrieve the team. Then we use a `temporary variable` to retrieve the same team.

1. Enter the following statement with a `pipe`:

```
nebula> GO FROM 100 OVER follow YIELD follow._dst AS id | \
        GO FROM $-.id OVER serve YIELD $$.team.name AS Team, \
        $^.player.name AS Player;
```

The following information is returned.

```
==============================
| Team    | Player            |
==============================
| Nuggets | Tony Parker       |
------------------------------
```

Here:

`$^` indicates the source vertex of the edge.

`|` indicates the pipe.

`$-` indicates the input flow. The output of the previous query `(id)` is used as the input of the next query `($-.id)`.

2. Enter the following statement with a `temporary variable`:

```
nebula> $var = GO FROM 100 OVER follow YIELD follow._dst AS id; \
        GO FROM $var.id OVER serve YIELD $$.team.name AS Team, \
        $^.player.name AS Player;
```

The following information is returned.

```
==============================
| Team    | Player            |
==============================
| Nuggets | Tony Parker       |
------------------------------
```

When a composition statement is submitted to the server as a whole, the custom variables in the statement have ended their life cycle.

EXAMPLE 4

If you have created and rebuilt an index for the data, you can use the `LOOKUP ON` keyword to query properties.

```
nebula> CREATE TAG INDEX player_index_0 on player(name);
nebula> REBUILD TAG INDEX player_index_0 OFFLINE;
```

Refer to the Index Doc to create an index.

The following example returns vertices whose name is `Tony Parker` and tagged with *player*.

```
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
YIELD player.name, player.age;
======================================
| VertexID | player.name | player.age |
======================================
| 101      | Tony Parker | 36         |
--------------------------------------
```

See LOOKUP Documentation for details.

EXAMPLE 5

The `SUBMIT JOB COMPACT` command triggers the long-term `RocksDB compact` operation. There are a lot of IO operations during the running of this command. After the command is executed successfully (the job status is stopped, see the example below), the read performance is improved.

For example:

```
nebula> SUBMIT JOB COMPACT;
==============
| New Job Id |
==============
```

```
| 40         |
--------------
```

The `SHOW JOBS` statement lists all the jobs that are not expired. For example:

```
nebula> SHOW JOBS;
=================================================================================
| Job Id | Command       | Status   | Start Time        | Stop Time         |
=================================================================================
| 22     | flush test2   | failed   | 12/06/19 14:46:22 | 12/06/19 14:46:22 |
---------------------------------------------------------------------------------
| 23     | compact test2 | stopped  | 12/06/19 15:07:09 | 12/06/19 15:07:33 |
---------------------------------------------------------------------------------
| 24     | compact test2 | stopped  | 12/06/19 15:07:11 | 12/06/19 15:07:20 |
---------------------------------------------------------------------------------
| 25     | compact test2 | stopped  | 12/06/19 15:07:13 | 12/06/19 15:07:24 |
---------------------------------------------------------------------------------
```

For more information about the `COMPACT` and `JOB` statements, refer to Job manager.

Batch Inserting

To insert multiple data, you can put all the DDL (Data Definition Language) statements in a `.ngql` file as follows.

```
CREATE SPACE basketballplayer(partition_num=10, replica_factor=1);
USE basketballplayer;
CREATE TAG player(name string, age int);
CREATE TAG team(name string);
CREATE EDGE follow(degree int);
CREATE EDGE serve(start_year int, end_year int);
```

- If you install **Nebula Graph** by compiling the source code, you can batch write to console by the following command:

```
$ cat schema.ngql | ./bin/nebula -u root -p nebula
```

- If you are using **Nebula Graph** by docker-compose, you can batch write to console by the following command:

```
$ cat basketballplayer.ngql | sudo docker run --rm -i --network=host \
vesoft/nebula-console:nightly --addr=127.0.0.1 --port=3699
```

Here:

- You must change the IP address and the port number to yours.
- You can download the `basketballplayer.ngql` file here.

Next to do

Before you start using Nebula Graph, make sure that you read the following documents. They include most of your questions about Nebula Graph. If you still have questions after reading these documents, please head over to our official forum to ask.

- Nebula Graph design and architecture
- Nebula Graph system requirement
- Vertex ID and partitions
- FAQ
- Nebula Graph Compact
- Nebula Graph video tutorial

Again, if you come across any problem following the steps in this guide, please head over to our official forum and our on-call developers are more than happy to answer your questions!

Finish the steps in this guide and feel Nebula Graph is good? Please star us on GitHub and make our day!

Last update: April 16, 2021

1.4.2 Frequently Asked Questions

Common questions about **Nebula Graph** and more. If you do not find the information you need in this documentation, please try searching the Users tab in the Nebula Graph official forum.

General Information

General Information lists the conceptual questions about **Nebula Graph**.

EXPLANATIONS ON THE TIME RETURN IN QUERIES

```
nebula> GO FROM 101 OVER follow
===============
| follow._dst |
===============
| 100         |
---------------
| 102         |
---------------
| 125         |
---------------
Got 3 rows (Time spent: 7431/10406 us)
```

Taking the above query as an example, the number `7431` in Time spent is the time spent by the database itself, that is, the time it takes for the query engine to receive a query from the console, fetch the data from the storage and perform a series of calculation ; the number `10406` is the time spent from the client's perspective, that is, the time it takes for the console from sending a request and receiving a response to displaying the result on the screen.

Trouble Shooting

Trouble Shooting session lists the common operation errors in **Nebula Graph**.

SERVER PARAMETER CONFIGURATION

In Nebula console, run

```
nebula> SHOW CONFIGS;
```

For configuration details, please see here.

HOW TO CHECK CONFIGS

Configuration files are stored under `/usr/local/nebula/etc/` by default.

UNBALANCED PARTITIONS

See Storage Balance.

LOG AND CHANGING LOG LEVELS

Logs are stored under `/usr/local/nebula/logs/` by default.

See graphd Logs and storaged Logs.

USING MULTIPLE HARD DISKS

Modify `/usr/local/nebula/etc/nebula-storage.conf` . For example

```
--data_path=/disk1/storage/,/disk2/storage/,/disk3/storage/
```

When multiple hard disks are used, multiple directories can be separated by commas, and each directory corresponds to a RocksDB instance for better concurrency. See here for details.

PROCESS CRASH

1. Check disk space `df -h` .

If there is not enough disk space, the service fails to write files, and crashes. Use the above command to check the current disk usage. Check whether the `--data_path` configured service directory is full.

2. Check memory usage `free -h`.

   The service uses too much memory and is killed by the system. Use `dmesg` to check whether there is an OOM record and whether there is keyword nebula in it.

3. Check logs.

ERRORS THROWN WHEN EXECUTING COMMAND IN DOCKER

This is likely caused by the inconsistency between the docker IP and the default listening address (172.17.0.2). Thus we need to change the the latter.

1. First run `ifconfig` in container to check your container IP, here we assume your IP is 172.17.0.3.
2. In directory `/usr/local/nebula/etc`, check the config locations of all the IP addresses with the command `grep "172.17.0.2" . -r`.
3. Change all the IPs you find in step 2 to your container IP 172.17.0.3.
4. Restart all the services.

ADDING TWO CLUSTERS ON A SINGLE HOST

When the same host is used for single host or cluster test, the storaged service cannot start normally. The listening port of the storaged service is red in the console.

Check the logs (/usr/local/nebula/nebula-storaged.ERROR) of the storaged service. If you find the "wrong cluster" error message, the possible cause is that the cluster id generated by **Nebula Graph** during the single host test and the cluster test are inconsistent. You need to delete the cluster.id file under the installation directory (/usr/local/nebula) and the data directory and restart the service.

CONNECTION REFUSED

```
E1121 04:49:34.563858   256 GraphClient.cpp:54] Thrift rpc call failed: AsyncSocketException: connect failed, type = Socket not open, errno = 111 (Connection refused): Connection refused
```

Check service status by using the following command:

```
$ /usr/local/nebula/scripts/nebula.service status all
```

Or list the hosts:

```
nebula> SHOW HOSTS;
```

Please make sure that the following ports are available and not blocked by your firewall. You can find them in the configuration files in `/usr/local/nebula/etc/`.

Service Name	Port
Meta Service	45500, 11000, 11002
Storage Service	44500, 12000, 12002
Graph Service	3699, 13000, 13002

Besides, remember to restart the services if you change the default value of the ports.

To check if the ports are available, run the following command:

```
$ telnet $ip $port
```

Make sure that the port in every host of the cluster is available.

COULD NOT CREATE LOGGING FILE:... TOO MANY OPEN FILES

1. Check your disk space `df -h`

2. Check log directory `/usr/local/nebula/logs/`
3. reset your max open files by `ulimit -n 65536`

HOW TO CHECK NEBULA GRAPH VERSION

Use the command `curl http://ip:port/status` to obtain the git_info_sha, the commitID of the binary package.

MODIFYING THE CONFIGURATION FILE DOES NOT TAKE EFFECT

Nebula Graph uses the following two methods obtaining configurations:

1. From the configuration files (You need to modify the files then restart the services);
2. From the Meta. Set via CLI and persists in Meta service. Please refer to the Configs Syntax for details.

Modifying the configuration file does not take effect because **Nebula Graph** gets configuration in the second method (from meta) by default. If you want to use the first way, please add the `--local_config=true` option in flag files `metad.conf`, `storaged.conf`, `graphd.conf` (flag files directory is `/home/user/nebula/build/install/etc`) respectively.

MODIFY ROCKSDB BLOCK CACHE

Modify the storage layer's configuration file `storaged.conf` (the default directory is `/usr/local/nebula/etc/`, yours maybe different) and restart the service. For example:

```
# Change rocksdb_block_cache to 1024 MB
--rocksdb_block_cache = 1024
# Stop storaged and restart
/usr/local/nebula/scripts/nebula.service stop storaged
/usr/local/nebula/scripts/nebula.service start storaged
```

Details see here.

NEBULA FAILS ON CENTOS 6.5

Nebula Graph fails on CentOS 6.5, the error message is as follows:

```
# storage log
Heartbeat failed, status:RPC failure in MetaClient: N6apache6thrift9transport19TTransportExceptionE: AsyncSocketException: connect failed, type = Socket not
open, errno = 111 (Connection refused): Connection refused

# meta log
Log line format: [IWEF]mmdd hh:mm:ss.uuuuuu threadid file:line] msg
E0415 22:32:38.944437 15532 AsyncServerSocket.cpp:762] failed to set SO_REUSEPORT on async server socket Protocol not available
E0415 22:32:38.945001 15510 ThriftServer.cpp:440] Got an exception while setting up the server: 92failed to bind to async server socket: [::]:0: Protocol not
available
E0415 22:32:38.945057 15510 RaftexService.cpp:90] Setup the Raftex Service failed, error: 92failed to bind to async server socket: [::]:0: Protocol not
available
E0415 22:32:38.949586 15463 NebulaStore.cpp:47] Start the raft service failed
E0415 22:32:38.949597 15463 MetaDaemon.cpp:88] Nebula store init failed
E0415 22:32:38.949796 15463 MetaDaemon.cpp:215] Init kv failed!
```

Nebula service status is as follows:

```
[root@redhat6 scripts]# ./nebula.service status  all
[WARN] The maximum files allowed to open might be too few: 1024
[INFO] nebula-metad: Exited
[INFO] nebula-graphd: Exited
[INFO] nebula-storaged: Running as 15547, Listening on 44500
```

Reason for error: CentOS 6.5 system kernel version is 2.6.32, which is less than 3.9. However, `SO_REUSEPORT` only supports Linux 3.9 and above.

Upgrading the system to CentOS 7.5 can solve the problem by itself.

THE PRECEDENCE BETWEEN MAX_EDGE_RETURNED_PER_VERTEX AND WHERE

If max_edge_returned_per_vertex is set to be 10, when filtering with WHERE, how many edges are returned if the actual edge number is greater than 10?

Nebula Graph performs WHERE condition filter first, if the actual number of edges is less than 10, then return the actual number of edges. Otherwise, it returns 10 edges according to the max_edge_returned_per_vertex value.

FETCH DOESN'T WORK SOMETIMES

When using `FETCH` to return data, sometimes the corresponding data is returned but sometimes it doesn't work.

If you encounter the above situation, please check if you have run two storage services on the same node and the port of the two storage service are the same. If so, please modify one of the storage port and import the data again.

DOES THE SST FILES SUPPORT MIGRATION

The sst files used by the storaged is bound to the graph space in the cluster. Thus you can not copy the sst files directly to a new graph space or a new cluster.

Last update: April 8, 2021

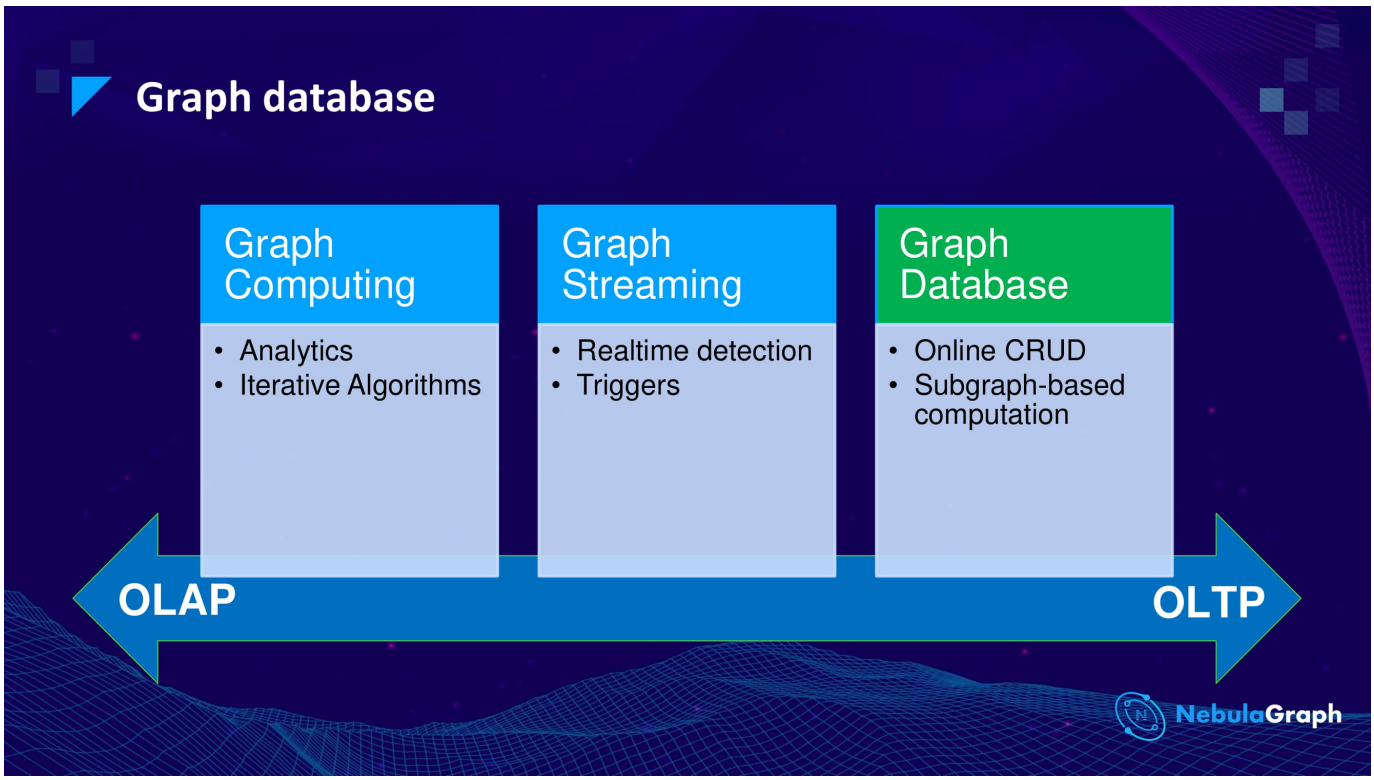
1.5 Design and Architecture

1.5.1 Design and Architecture of Nebula Graph

This document is to walk you through on how **Nebula Graph** is designed and why, and it will be separated into two parts: Industry Overview and **Nebula Graph** Architecture.

Industry Overview

OLAP & OLTP IN GRAPH



The axis in the above picture shows the different requirements for query latency. Like a traditional database, graph database can be divided into two parts: OLAP and OLTP. OLAP cares more about offline analysis while OLTP prefers online processing. Graph computing framework in OLAP is used to analyse data based on graph structure. And it's similar to OLAP in traditional database. But it has features which are not available in traditional database, one is iterative algorithm based on graph. A typical example is the PageRank algorithm from Google, which obtains the relevance of web pages through constant iterative computing. Another example is the commonly-used LPA algorithm.

Along the axis to right, there comes the graph streaming field, which is the combination of basic computing and streaming computing. A relational network is not a static structure, rather, it constantly changes in the business: be it graph structure or graph properties. Computing in this filed is often triggered by events and its latency is in second.

Right beside the graph streaming is the online response system, whose requirement for latency is extremely high, which should be in millisecond.

The rightmost field is graph database and its use cases are completely different from the one on the left. The **Nebula Graph** we're working on can find its usage here in OLTP.

NATIVE VS MULTI-MODEL

Graph databases can be classified into two kinds: native graph database and multi-model graph database. Using a `graph first` design, native graph databases are specifically optimized in storage and processing, thus they tend to perform queries faster,

scale bigger and run more efficiently, calling for much less hardware at the same time. As for multi-model products, their storage comes from an outside source, such as a relational, columnar, or other NoSQL database. These databases use other algorithms to store data about vertices and edges and can lead to latent results as their storage layer is not optimized for graphs.

DATA STORED IN GRAPH DATABASE

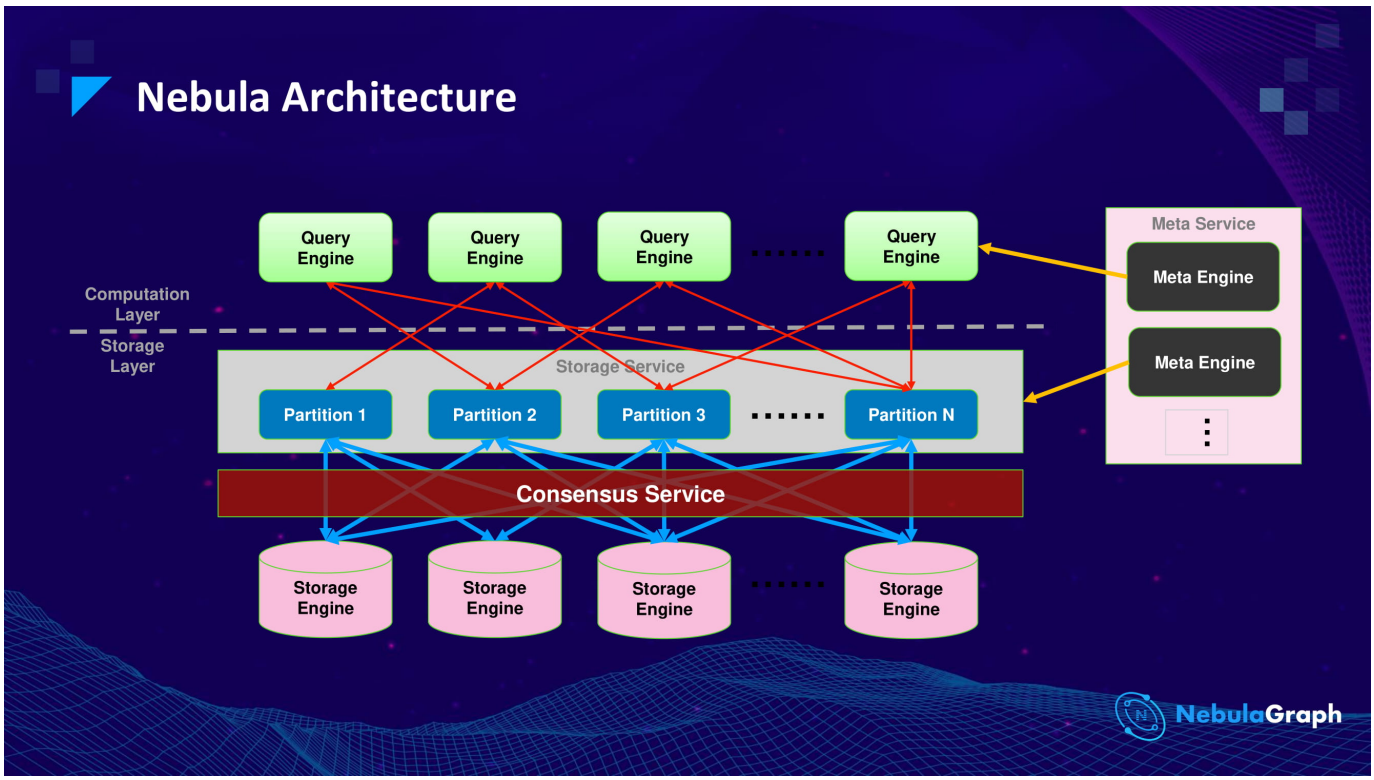
In graph database, data is stored as graph. Modelling data as graph is natural, and has the nice benefit of staying legible even by non-technical people. The data model handled by **Nebula Graph** is ***directed property graph***, whose edges are directional and there could be properties on both edges and vertices. It can be represented as:

$$G = < V, E, P_V, P_E >$$

Here **V** is a set of nodes, aka vertices, **E** is a set of directional edges, P_V represents properties on vertices, and P_E is the properties on edges.

Nebula Graph Architecture

Designed based on the above features, **Nebula Graph** is an open source, distributed, lightning-fast graph database, it is composed of four components: storage service, meta service, query engine and client.



The dashed line in the above picture divided computing and storage as two independent parts, the upper is the computing service, each machine or virtual machine is stateless and never talks to other so it's easy to scale in or out; the lower is the storage service, it's stateful since data is stored there. Storage service can turn graph semantics into key-values and pass them to the KV-store below it. Between the two is the Raft protocol.

The right side is the meta service, similar to the NameNode in HDFS, it stores all metadata like schema and controls scaling.

DESIGN THINKING: STORAGE SERVICE

Nebula Graph adopted the **shared-nothing** distributed architecture in storage so nodes do not share memory or storage, which means there are no central nodes in the whole system. Benefits of such design are:

- Easy to scale
- The overall system continues operating despite individual crash

Another design is the **separation of computing and storage**, and the benefits are as follows:

- Scalability. Separating storage from computing makes storage service flexible, thus it's easy to scale out or in.
- Availability. Recovery from vertex failure can be performed quickly.

The binary of storage service is **nebula-storaged**, which provides a key-value store. Multiple storage engines like RocksDB and HBase are supported, with RocksDB set as the default engine. To build a resilient distributed system, Raft is implemented as the consensus algorithm.

Raft achieves data consensus via an elected leader. Based on that, nebula-storaged makes the following optimizations:

- Parallel Raft

Partitions of the same ID from multiple machines form a raft group. And the parallel operations are implemented with multiple sets of Raft groups.

- Write Path & batch

In Raft protocol, the master replicates log entries to all the followers and commits the entries in order. To improve the write throughput, **Nebula Graph** not only employs the parallel raft, but also implements the dynamic batch replication.

- Load-balance

Migrating the partitions on an overworked server to other relatively idle servers to increases availability and capacity of the system.

DESIGN-THINKING: META SERVICE

The binary of the meta service is **nebula-metad**. Here is the list of its primary functionalities:

- User management

  In **Nebula Graph** different roles are assigned diverse privileges. We provide the following native roles: Global Admin, Graph Space Admin, User and Guest. - Cluster configuration management

  Meta service manages the servers and partitions in the cluster, e.g. records location of the partitions, receives heartbeat from servers, etc. It balances the partitions and manages the communication traffic in case of server failure. - Graph space management

  Nebula Graph supports multiple graph spaces. Data in different graph spaces are physically isolated. Meta service stores the metadata of all spaces in the cluster and tracks changes that take place in these spaces, like adding, dropping space, modifying graph space configuration (Raft copies). - Schema management

  Nebula Graph is a strong typed database.

  - Types of tag and edge properties are recorded by meta service. Supported data types are: int, double, timestamp, list, etc.
  - Multi-version management, supporting adding, modifying and deleting schema, and recording its version.
  - TTL (time-to-live) management, supporting automatic data deletion and space reclamation. The meta service is stateful, and just like the storage service, it persists data to a key-value store.

DESIGN-THINKING: QUERY ENGINE

Nebula Graph's query language **nGQL** is a SQL-like descriptive language rather than an imperative one. It's compossible but not embeddable, it uses Shell pipe as an alternative, aka output in the former query acts as the input in the latter one. Key features of nGQL are as follows:

- Primary algorithms are built in the query engine
- Duplicate queries can be avoided by supporting user-defined function (UDF)
- Programmable

The binary of the query engine is **nebula-graphd**. Each nebula-graphd instance is stateless and never talks to other nebula-graphd. nebula-graphd only talks to the storage service and the meta service. That makes it trivial to expand or shrink the query engine cluster.

The query engine accepts the message from the client and generates the execution plan after the lexical parsing (Lexer), semantic analysis (Parser) and the query optimization. Then the execution plan will be passed to the execution engine. The query execution engine takes the query plans and interacts with meta server and the storage engine to retrieve the schema and data.

The primary optimizations of the query engine are:

- Asynchronous and parallel execution

  I/O operations and network transmission are time-consuming. Thus asynchronous and parallel operations are widely adopted in the query engine to reduce the latency and to improve the overall throughput. Also, a separate resource pool is set for each query to avoid the long-tail effect of those time-consuming queries.

- Pushing down computation

  In a distributed system, transferring a large amount of data on the network really extends the overall latency. In **Nebula Graph**, the query engine will make decisions to push some filter and aggregation down to the storage service. The purpose is to reduce the amount of data passing back from the storage.

DESIGN-THINKING: API AND SDK

Nebula Graph provides SDKs in C++, Java, and Golang. **Nebula Graph** uses fbthrift as the RPC framework to communicate among servers. **Nebula Graph**'s web console is in progress and will be released soon.

Last update: April 8, 2021

1.5.2 Storage Design

Abstract

This document gives an introduction to the storage design of the graph database **Nebula Graph**.

The Storage Service of **Nebula Graph** is composed of two parts. One is `Meta Service` that stores the meta data, the other is `Storage Service` that stores the data. The two services are in two independent processes. The data directory and deployment are separated but their architectures are almost the same.

Architecture

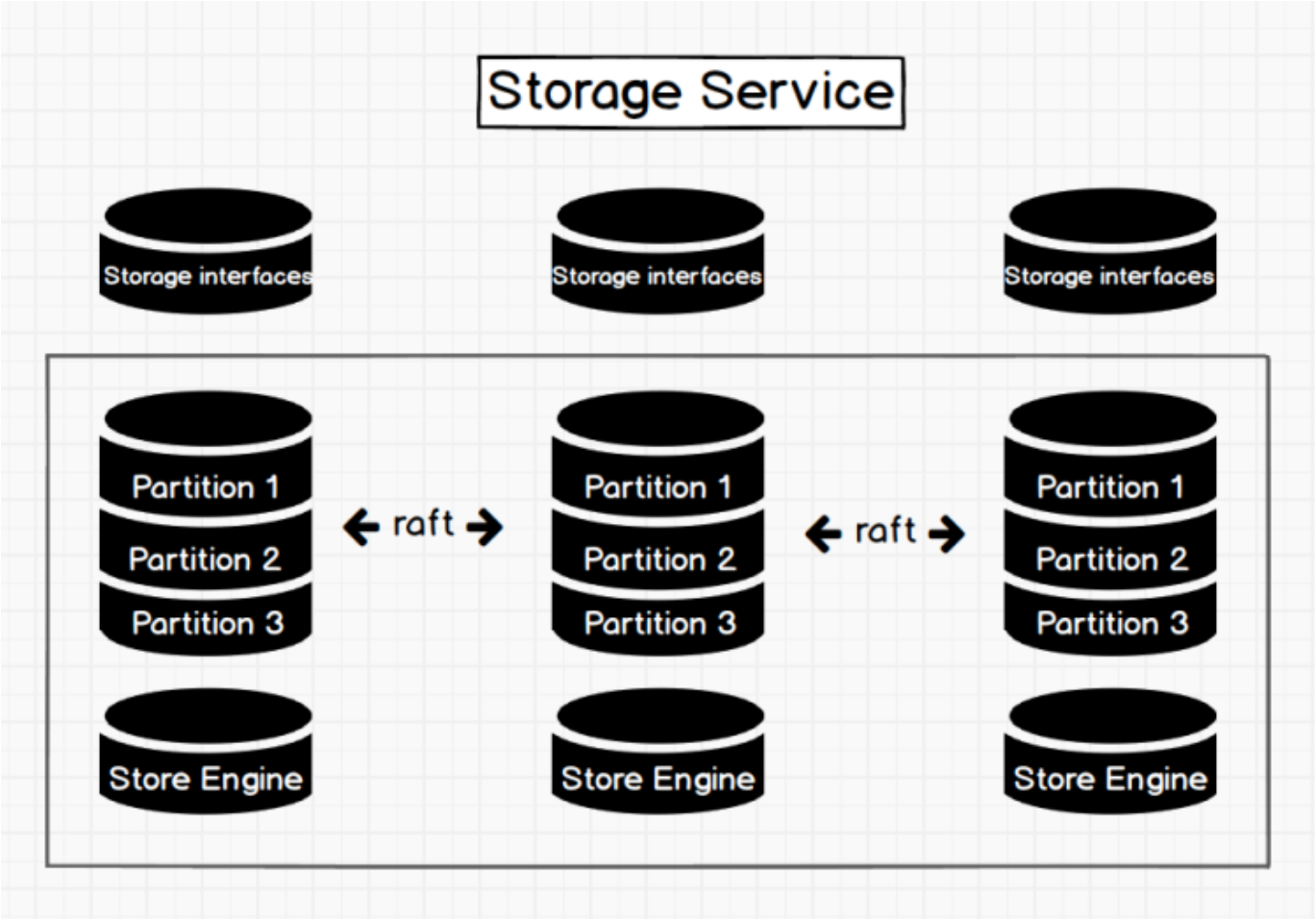


Fig. 1 The Architecture of Storage Service

As shown in Fig. 1, there are three layers in Storage Service. The bottom layer is the local storage engine, providing `get`, `put`, `scan` and `delete` operations on local data. The related interfaces are in `KVStore/KVEngine.h` and users can develop their own local store plugins based on their needs. Currently, **Nebula Graph** provides store engine based on RocksDB.

Above the local storage engine is the consensus layer that implements multi group raft. Each partition corresponding to a Raft group, is for the data sharding. Currently, **Nebula Graph** uses `hash` to shard data. When creating a space, users need to specify the partition number. Once set, partition number cannot be changed. Generally, the partition number must meet the need to scale-out in the future.

Above the consensus layer is the storage interface that defines a set of APIs that are related to graph. These API requests are translated into a set of kv operations to the corresponding partition. It is this layer that makes the storage service a real graph storage, otherwise it's just a kv storage. **Nebula Graph** doesn't use kv-store as an independent service as a graph query involves a lot of calculation that involves schema, which is not existed in the kv layer. Such architecture makes computation operation pushing down more easily.

Schema & Partition

As a graph database, **Nebula Graph** stores the vertices, edges and their properties. How to efficiently filtering or projecting is critical for a graph exploration.

Nebula Graph uses **tags** to indicate a vertex type. One vertex can have multiple types (and therefore multiple tags), and each tag defines its own properties. In the kv store, we use `vertex_ID + Tag_ID` together as a key, and the corresponding value are the encoded property. The format is shown in Fig. 2:

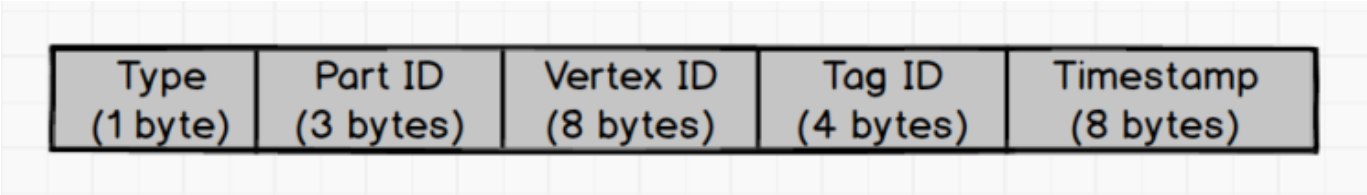


Fig. 2 Vertex Key Format

- `Type`: one byte, to indicate the key type. e.g. data, index, system, etc.
- `Part ID`: three bytes, used to indicate the (sharding) partition id. It's designed for the data migration/balance operation by **prefix-scanning all the data in a partition.**
- `Vertex ID`: eight bytes, used to indicate vertex ID. Two vertices with an identity vertexID are considered as the same one.
- `Tag ID`: four bytes, used to indicate its tag's (encoded) ID.
- `Timestamp`: eight bytes, not visible to users. Reserved for `Multiversion concurrency control (MVCC)`.

Each edge in **Nebula Graph** is modeled and stored as two independent key-values. One, namely the `out-edge`, is stored in the same partition as the `source vertex`. The other one, namely `in-edge`, is stored in the same partition as the `destination vertex`. So generally, out-key and in-key are in different partitions.

Between two vertices, edges with the same type are acceptable, and different types are legal as well. For example, by defining an edge type '`money-transfer-to`', `user A` can transfer money to `user B` at two timestamps. Thus a field, namely `rank`, is added to (the key part of the timestamp to) distinguish which transfer records is referring. Edge key format is shown in Fig. 3:

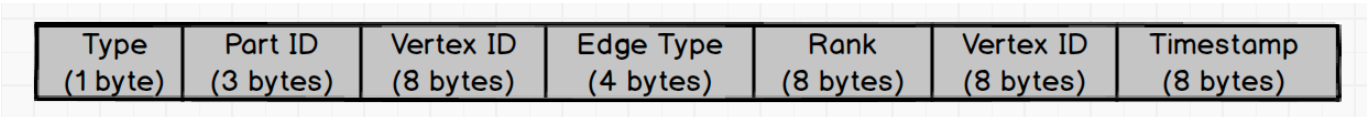


Fig. 3 Edge Key Format

- `Type`: one byte, used to indicate key type. E.g., data, index, system, etc.
- `Part ID`: three bytes. The same as in Fig. 2.
- `Vertex ID`: eight bytes, used to indicate `source vertex ID` of an `out-edge` (Fig. 4), and `destination vertex ID` of an `in-edge` (Fig. 5). See below.
- `Edge Type`: four bytes, used to indicate (encoded) edge type id. A positive number means that this key is an `out-edge`, and a negative number indicates that this is an `in-edge`.
- `Rank`: eight bytes, used in multiple edges with the same type. E.g., It can store *transaction time*, *transaction amount*, or *edge weight*.
- `Timestamp`: eight bytes. The same as in Fig. 2.

If `Edge Type` is positive, the corresponding edge key format is shown in Fig. 4; otherwise, the corresponding edge key format is shown in Fig. 5.

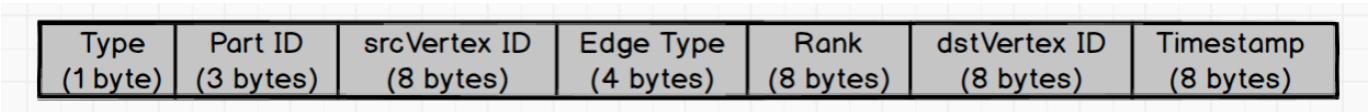


Fig. 4 Out-key format

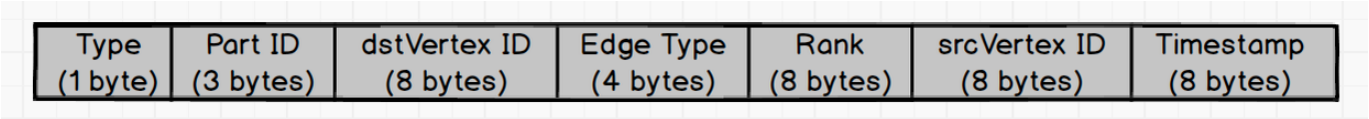


Fig. 5 In-key format

Besides the key part above, the value part is the encoded properties (of a vertex or an edge). As a strong typed database, **Nebula Graph** gets the schema information from the Meta Service before encoding/decoding. And multi-version schema are also considered when altering schema.

Nebula Graph shards data through `modulo operation` on `vertex ID`. All the *out-keys*, *in-keys* and *tag id* are placed in the same partition. This improves query efficiency as a local/non-remote file access. Breadth-First-Search (BFS) expansion starting from a given vertex is a very common ad-hoc graph exploration. And during BFS, the performance of filtering out edge/vertex properties are time-consuming. **Nebula Graph** guarantees the operation efficiency by putting properties of a vertex and its edges locating near each other. It is worth noting that most graph databases vendors run their benchmarks with `Graph 500` or `Twitter` data set, which are of no eloquence because the properties are not taken into consideration in this kind of graph exploration. While most production cases are not that simple.

KVStore

Nebula Graph writes its own kv store to meet the performance needs:

- **High performance**, a pure high performance key value store.
- **Provided as a library**, as a strong typed database, the performance of storage layer is key to **Nebula Graph**.
- **Strong data consistency**, since **Nebula Graph** is a distribution system.
- **Written in C++**, as most of our developers are C++ programers.

For users who are not sensitive to performance or unwilling to migrate data from other storage systems, such as HBase or MySQL, **Nebula Graph** also provides a plugin over the kv store to replace its default RocksDB. Currently, HBase plugin has been released yet.

As RocksDB is the local storage engine, **Nebula Graph** can manage multiple hard disks to take full use of the parallel IO access. What a user needs to do is to configure multiple data directories.

Nebula Graph manages the distributed kv store in with meta service. All the partition distribution and cluster machine status can be found in the meta service. Users can input commands in the console to add or remove machines to generate and execute a balance plan in meta service.

Nebula Graph writes its own (Write-Ahead-Log, WAL) module to replace the default one in RocksDB. Since the WAL is used for (distributed system's) Raft consensus. Each partition has a WAL, so after a (crash and) reboot, the partition can catch up its own data, and there is no need to split WAL between several partitions.

Besides, **Nebula Graph** defines a special category, namely `Command Operation Log`, to conduct some command operations. These logs are very short, with no real data, and are only used to inform all replicas to execute certain command operations with raft protocol. What's more, since the logs are serialized in the Raft protocol, **Nebula Graph** also provides another class, namely `Atomic Operation Log`, to conduct the atomic operation between the replicas of a partitions. E.g., the compare-and-set (CAS) or read-modify-write operations are atomic in **Nebula Graph** per partition.

A **Nebula Graph** cluster can have multiple individual graph spaces. Each space has its own partition number and replica copies. Different spaces are isolated physically from each other in the same cluster. Besides, the spaces can also have very different storage engines and sharding strategies. E.g., One space can use HBase as its storage backend with alphabet ranging sharding, and the other space uses the default RocksDB with hashing sharding. And these two spaces are running in the same **Nebula Graph** cluster.

Raft Implementation

This part gives some details on how the raft protocol is implemented in **Nebula Graph**.

MULTI RAFT GROUP

According to Raft requirement, the log ID must be in a sequential order. Therefore, almost all the raft implementations will use `Multi Raft Group` to increase the concurrency. Therefore, the number of partition will determine how many operations can be executed simultaneously. But you can not simply add too much partitions in the system, which can have some side affects. Each raft group stores many state information and (as mentioned earlier) it has a WAL file. Thus, the more partitions, the more footprint costs. Also, if the work load is low, the batch operation can not gain from the parallel. E.g., consider a system with ten thousand partitions. For every second, there are about ten thousands write-in requests. You can calculate that in average, for every partition, there is only one write-in request. So from the client side, it's a 100k batch write. But from the partition side, it's a single write.

There are two key challenges to implement the Multi Raft Group. **First one is how to share the transport layer**. Because each Raft Group sends messages to its corresponding peers, if the transport layer cannot be shared, the connection costs will be very high. **Second one is how to design the multi-threading model**. Raft Groups share the same thread pool to prevent starting too many threads and a high context switch cost.

BATCH

For each Partition, it is necessary to do batch multiple operations together to improve throughput when writing WAL serially. In general, there is nothing special about batch, but **Nebula Graph** designs some special types of WAL based on each part serialization, which brings some challenges.

For example, **Nebula Graph** uses WAL to implement lock-free CAS operations. And every CAS operation will be executed until the previous WAL has been committed. So for a batch, if there are some logs contain CAS operation, we need to divide this batch into several smaller (sub)groups. And make sure these (sub)groups are executed in sequential order.

LEARNER

When a new machine is added to a cluster, it has to catch up data for quite a long time. And there may be accidents during this process. If this one directly joins the raft group as a follower role, it will dramatically reduce the availability of the entire cluster. **Nebula Graph** introduces the learner role, and it is implemented by the command WAL mentioned above. When a leader is writing WAL and meets an `add learner command`, it will add the new coming-in learner to its peers list and mark it as a learner. The logs will send to all the peers, both the followers and the learner. But the learner can not vote for the leader's election.

TRANSFER LEADERSHIP

Transfer leadership is extremely important during a data balance operation. When migrating a partition from one machine to another, **Nebula Graph** will first check if it is a leader. If so, another follower should be elected as a leader before the migration. Otherwise, the cluster service is affected since the leader is on migration. After the migration is done, a `BALANCE LEADER` command is invoked, so that the work load on each machine can be balanced.

When transferring leadership, it is worth noting the timing when a leader abandons the leadership and when all the followers start a leader election. When a `transfer leadership` command is committed, from the leader's view, it loses the leadership. From other followers' view, when receiving this command, it starts a new leader election. These two operations must be executed in the same process with a normal raft leader election. Otherwise, some corner cases can occur and they are very hard to test.

MEMBERSHIP CHANGE

To avoid the brain-split, when Raft Group members changed, an intermediate state is required. In such state, the majority of the old group and new group always have an overlap. This majority overlap will prevent neither group from making decisions unilaterally. This is the `joint consensus` as mentioned in the famous Raft thesis. To make it even simpler, Diego Ongaro suggests to **add or remove only one peer at a time to ensure the overlap between the majority** in his doctoral thesis. **Nebula Graph**'s implementation also uses this approach, except that the implementation to add or remove member is different. For details, please refer to `addPeer/removePeer` in Raft Part source code.

SNAPSHOT

Take snapshot is a common command during daily DBA operations. But snapshot operation will introduce extra challenges when considering together with the raft protocol. It's very error-prone.

E.g., what if the leader loses its leadership in an election when sending a snapshot command. What should we do. In this situation, the follower may only receive half log of the snapshot command, should we cleanup and rollback? Because multiple partitions share a single storage, how to clean up the data is a cumbersome work. In addition, the snapshot process will start a heavy write to disks. To avoid slow down the frontend reads and writes, we do not want snapshot process to share the same IO threadPool with the normal Raft logs. Besides, snapshot also requires large footprint, which is critical for online service performance.

Storage Service

The Interfaces of Storage Service layer are:

- `Insert vertex/edge`: insert a vertex or edge and its properties.
- `getNeighbors`: get the in-edge or out-edge from a set of vertices. And return the edges and properties. Condition filtering are also considered.
- `getProps`: get the properties of a vertex or an edge.

Graph semantics interfaces are translated into kv operations in this layer as well. In order to improve the performance, concurrent operations are also implemented in this layer.

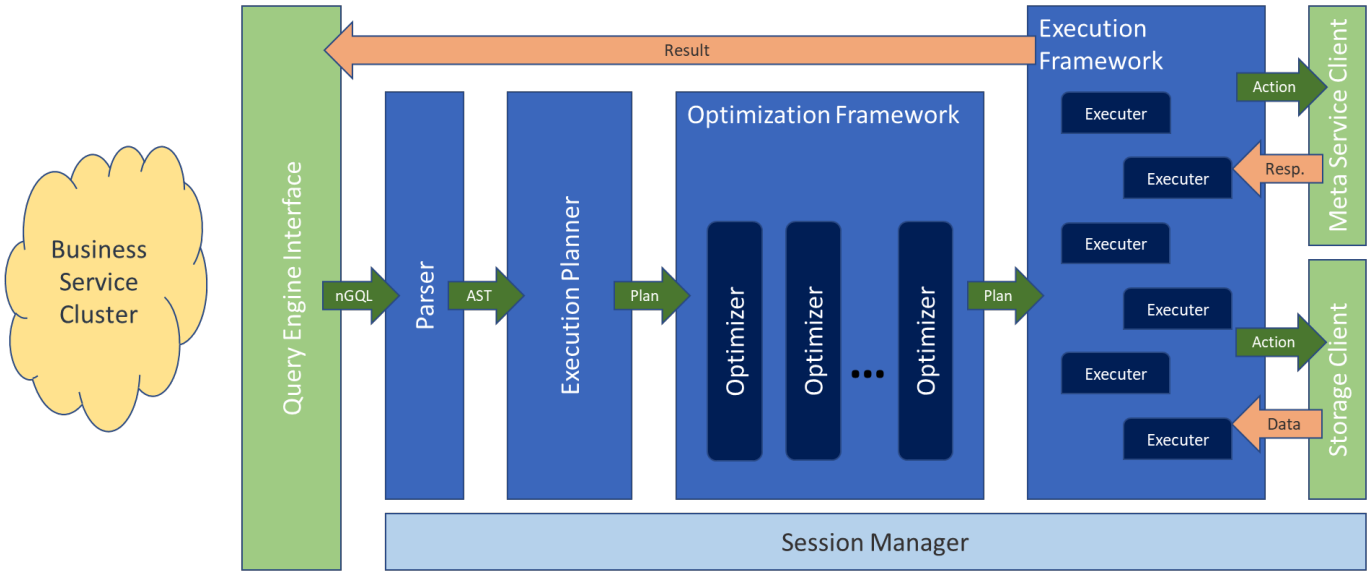
Meta Service

Nebula Graph wrap up a set of meta-related interfaces from the kv store interface (as mentioned earlier). Meta service can support CRUD operation on schema, cluster administration and user privileges. Meta service can be deployed on a single host, but it is recommended to deploy on multiple hosts with at least three or five replicas to get a better availability and fault tolerance.

Last update: April 8, 2021

1.5.3 Query Engine Overview

The query engine is used to process the **Nebula Graph** Query Language (nGQL) statements. This article gives an architectural overview of the **Nebula Graph** query engine.



Above is the overview chart of the query engine. If you are familiar with the SQL execution engine, this should be no stranger to you. In fact, the **Nebula Graph** query engine is very similar to the modern SQL execution engine except the query language parser and the real actions in the execution plan.

Session Manager

Nebula Graph employs the Role Based Access Control. So when the client first connects to the Query Engine, it needs to authenticate itself. When it succeeds, the query engine creates a new session and returns the session ID to the client. All sessions are managed by the Session Manager. The session will remember the current graph space and the access rights to the space. The session will keep some session-wide configurations and be used as an temporary storage to store information across multiple requests in the same session as well.

The session will be dropped when the client connection is closed, or being idle for a period of time. The length of the idle time is configurable.

When the client sends a request to the query engine, it needs to attach the current session ID, otherwise, the query engine will reject the request.

When the query engine accesses the storage engines, it will attach the session object to every request, so that the storage engine does not have to manage sessions.

Parser

The first thing that the query engine will do when receiving a request is to parse the statements in the request. This is done by the parser. The majority of the parser code is generated by the famous flex/bison tool set. The lexicon and syntax files for the nGQL can be found in the `src/parser` folder in the source code tree. The nGQL is designed in a way that is close enough to SQL. The idea is to smoothen the learning curve as much as possible.

Graph databases currently do not have a unified international query language standard. As soon as the ISO's GQL committee releases their first draft, we will make nGQL compatible with the proposed GQL.

The output of the parser is an Abstract Syntax Tree (AST), which will be passed on to the next module: Execution Planner.

Execution Planner

The Execution Planner will convert the AST from the parser into a list of actions (execution plan). An action is the smallest unit that can be executed. A typical action could be fetching all neighbors for a given vertex, getting properties for an edge, or filtering vertices or edges based on the given condition.

When converting AST to the execution plan, all IDs will be extracted, so that the execution plan can be reused. The extracted IDs will be placed in the context for the current request. The context will be used to store the variables and intermediate results as well.

Optimization

The newly generated execution plan will then be passed to the Optimization Framework. There are multiple optimizers registered in the framework. The execution plan will be passed through all optimizers sequentially. Each optimizer has the opportunity to modify (optimize) the plan.

At the end, the final plan can look dramatically different from the original plan, but the execution result should be exactly same as the original plan.

Execution

The last step in the query engine is to execute the optimized execution plan. It's done by the Execution Framework. Each executor will process one execution plan at a time. Actions in the plan will be executed one by one. The executor will do limited local optimization as well, such as deciding whether to run in parallel.

Depending on the action, the executor will communicate with the Meta Service or the Storage Engine via their clients.

Last update: April 8, 2021

2. Query Language

2.1 Reader

This chapter is for those who want to use `Nebula Graph` query language.

2.1.1 Example Data

The example data used in **Nebula Graph** query statements can be downloaded here. After downloading the example data, you can import it to your **Nebula Graph** database with Nebula Graph Studio.

2.1.2 Placeholder Identifiers and Values

The query language of **Nebula Graph** is nGQL. Refer to the following standards in nGQL:

- ISO/IEC 10646
- ISO/IEC 39075
- ISO/IEC NP 39075 (Draft)

In template code, any token that is not a keyword, a literal value, or punctuation is a placeholder identifier or a placeholder value.

For details of the symbols in nGQL, refer to the following table:

Token	Meaning
< >	name of a syntactic element
::=	formula that defines an element
[ ]	optional elements
{ }	explicitly specified elements
|	complete alternative elements
...	may be repeated any number of times

Last update: April 8, 2021

2.2 Data Types

2.2.1 Data Types

The built-in data types supported by **Nebula Graph** are as follows:

Numeric Types

INTEGER

An integer is declared with keyword `int` , which is 64-bit *signed*, the range is [-9223372036854775808, 9223372036854775807]. Integer constants support multiple formats:

1. Decimal, for example `123456`
2. Hexadecimal, for example `0xdeadbeaf`
3. Octal, for example `01234567`

DOUBLE FLOATING POINT

Double floating point data type is used for storing double precision floating point values. Keyword used for double floating point data type is `double`. There are no upper and lower ranges.

Boolean

A boolean data type is declared with the `bool` keyword and can only take the values `true` or `false`.

String

The string type is used to store a sequence of characters (text). The literal constant is a sequence of characters of any length surrounded by double or single quotes. Line breaks are not allowed in a string. For example `"Shaquile O'Neal"`,`'"This is a double-quoted literal string"'`. Embedded escape sequences are supported within strings, for example:

- `"\n\t\r\b\f"`
- `"\110ello world"`

Timestamp

- The supported range of timestamp type is '1970-01-01 00:00:01' UTC to '2262-04-11 23:47:16' UTC
- Timestamp is measured in units of seconds
- Supported data inserting methods
  - call function now()
  - Time string, for example: "2019-10-01 10:00:00"
  - Input the timestamp directly, namely the number of seconds from 1970-01-01 00:00:00
- When the inserted timestamp is string, **Nebula Graph** converts TIMESTAMP values from the current time zone to **UTC** for storage, and back from UTC to the **current time** zone for retrieval
- The underlying storage data type is: **int64**

Examples

Create a tag named school

```
nebula> CREATE TAG school(name string , create_time timestamp);
```

Insert a vertex named "stanford" with the foundation date "1885-10-01 08:00:00"

```
nebula> INSERT VERTEX school(name, create_time) VALUES hash("new"):("new", "1985-10-01 08:00:00")
```

Insert a vertex named "dut" with the foundation date now

```
nebula> INSERT VERTEX school(name, create_time) VALUES hash("dut"):("dut", now())
```

Last update: April 8, 2021

2.2.2 Type Conversion

Converting an expression of a given type to another type is known as type-conversion. In nGQL, type conversion is divided into implicit conversion and explicit conversion.

Implicit Type Conversion

Implicit conversions are automatically performed when a value is copied to a compatible type.

1. Following types can implicitly converted to `bool`:
   - The conversions from/to bool consider `false` equivalent to `0` for empty string types, `true` is equivalent to all other values.
   - The conversions from/to bool consider `false` equivalent to `0` for int types, `true` is equivalent to all other values.
   - The conversions from/to bool consider `false` equivalent to `0.0` for float types, `true` is equivalent to all other values.
2. `int` can implicitly converted to `double`.

Explicit Type Conversion

In addition to implicit type conversion, explicit type conversion is also supported in case of semantics compliance. The syntax is similar to the `C` language:

`(type_name)expression`.

For example, the results of `YIELD length((string)(123)), (int)"123" + 1` are `3, 124` respectively. The results of `YIELD (int)(TRUE)` is `1`. And `YIELD (int)("12ab3")` fails in conversion.

Last update: April 8, 2021

2.3 Functions and Operators

2.3.1 Bitwise Operators

Name	*Description*
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR (XOR)

Last update: April 8, 2021

2.3.2 Built-in Functions

Nebula Graph supports calling built-in functions of the following types:

Math

Function	Description
double abs(double x)	Return absolute value of the argument
double floor(double x)	Return the largest integer value smaller than or equal to the argument. (Rounds down)
double ceil(double x)	Return the smallest integer greater than or equal to the argument. (Rounds up)
double round(double x)	Return integral value nearest to the argument, returns a number farther away from 0 if the parameter is in the middle
double sqrt(double x)	Return the square root of the argument
double cbrt(double x)	Return the cubic root of the argument
double hypot(double x, double y)	Return the hypotenuse of a right-angled triangle
double pow(double x, double y)	Compute the power of the argument
double exp(double x)	Return the value of e raised to the x power
double exp2(double x)	Return 2 raised to the argument
double log(double x)	Return natural logarithm of the argument
double log2(double x)	Return the base-2 logarithm of the argument
double log10(double x)	Return the base-10 logarithm of the argument
double sin(double x)	Return sine of the argument
double asin(double x)	Return inverse sine of the argument
double cos(double x)	Return cosine of the argument
double acos(double x)	Return inverse cosine of the argument
double tan(double x)	Return tangent of the argument
double atan(double x)	Return inverse tangent the argument
int rand32()	Return a random 32 bit integer
int rand32(int max)	Return a random 32 bit integer in [0, max)
int rand32(int min, int max)	Return a random 32 bit integer in [min, max)
int rand64()	Return a random 64 bit integer
int rand64(int max)	Return a random 64 bit integer in [0, max)
int rand64(int min, int max)	Return a random 64 bit integer in [min, max)

String

NOTE: Like SQL, nGQL's character index (location) starts at `1`, not like C language from `0`.

Function	Description
int strcasecmp(string a, string b)	Compare strings without case sensitivity, when a = b, return 0, when a > b returned value is greater than 0, otherwise less than 0
string lower(string a)	Return the argument in lowercase
string upper(string a)	Return the argument in uppercase
int length(string a)	Return length (int) of given string in bytes
string trim(string a)	Remove leading and trailing spaces
string ltrim(string a)	Remove leading spaces
string rtrim(string a)	Remove trailing spaces
string left(string a, int count)	Return the substring in [1, count], if length a is less than count, return a
string right(string a, int count)	Return the substring in [size - count + 1, size], if length a is less than count, return a
string lpad(string a, int size, string letters)	Left-pads a string with another string to a certain length
string rpad(string a, int size, string letters)	Reft-pads a string with another string to a certain length
string substr(string a, int pos, int count)	Extract a substring from a string, starting at the specified position, extract the specified length characters
int hash(string a)	Encode the data into integer value

Explanations on the returns of function `substr`:

- If pos is 0, return empty string
- If the absolute value of pos is greater than the string, return empty string
- If pos is greater than 0, return substring in [pos, pos + count)
- If pos is less than 0, and set position N as length(a) + pos + 1, return substring in [N, N + count)
- If count is greater than length(a), return the whole string

Timestamp

Function	Description
int now()	Return the current date and time

Last update: April 8, 2021

2.3.3 Comparison Functions and Operators

Name	Description
=	Assign a value
/	Division operator
==	Equal operator
!=	Not equal operator
<	Less than operator
<=	Less than or equal operator
-	Minus operator
%	Modulo operator
+	Addition operator
*	Multiplication operator
-	Change the sign of the argument
udf_is_in()	Whether a value is within a set of values

Comparison operations result in a value of *true* and *false*.

- ==

Equal. String comparisons are case-sensitive. Values of different types are not equal.

```
nebula> YIELD 'A' == 'a';
==============
| ("A"=="a") |
==============
| false |
--------------

nebula> YIELD '2' == 2;
[ERROR (-8)]: A string type can not be compared with a non-string type.
```

- >

Greater than

```
nebula> YIELD 3 > 2;
=========
| (3>2) |
=========
| true |
---------
```

- ≥

Greater than or equal to:

```
nebula> YIELD 2 >= 2;
[ERROR (-8)]: A string type can not be compared with a non-string type.
```

- <

Less than:

```
nebula> YIELD 2.0 < 1.9;
=========================================
| (2.000000000000000<1.900000000000000) |
=========================================
| false                                 |
-----------------------------------------
```

- ≤

Less than or equal to:

```
nebula> YIELD 0.11 <= 0.11;
========================
| (0.110000<=0.110000) |
========================
|true |
------------------------
```

- !=

Not equal:

```
nebula> YIELD 1 != '1';
A string type can not be compared with a non-string type.
```

- udf_is_in()

Returns true if the first value is equal to any of the values in the list, otherwise, returns false.

```
nebula> YIELD udf_is_in(1,0,1,2);
======================
| udf_is_in(1,0,1,2) |
======================
| true               |
----------------------

nebula> GO FROM 100 OVER follow WHERE udf_is_in($$.player.name, "Tony Parker"); /* This example might not work because udf_is_in might be changed in the
future.*/
===============
| follow._dst |
===============
| 101         |
---------------

nebula> GO FROM 100 OVER follow YIELD follow._dst AS id | GO FROM $-.id OVER follow WHERE udf_is_in($-.id, 102, 102 + 1);
===============
| follow._dst |
===============
| 100         |
---------------
| 101         |
---------------
```

Last update: April 8, 2021

2.3.4 Group By

`GROUP BY` is similar with SQL. It can only be applied in the `YIELD` syntax.

Name	Description
AVG()	Return the average value of the argument
COUNT()	Return the number of records
COUNT_DISTINCT())	Return the number of different values
MAX()	Return the maximum value
MIN()	Return the minimum value
STD()	Return the population standard deviation
SUM()	Return the sum
BIT_AND()	Bitwise AND
BIT_OR()	Bitwise OR
BIT_XOR()	Bitwise exclusive OR (XOR)

All the functions above only work with int64 and double.

Example

```
nebula> GO FROM 100 OVER follow YIELD $$.player.name as Name | GROUP BY $-.Name YIELD $-.Name, COUNT(*);
-- Find all the players followed by vertex 100 and return their names as Name. These players are grouped by Name and the number in each group is counted.
-- The following result is returned:
================================
| $-.Name           | COUNT(*) |
================================
| Kyle Anderson     | 1        |
--------------------------------
| Tony Parker       | 1        |
--------------------------------
| LaMarcus Aldridge | 1        |
--------------------------------

nebula> GO FROM 101 OVER follow YIELD follow._src AS player, follow.degree AS degree | GROUP BY $-.player YIELD SUM($-.degree);
-- Find all the players followed by vertex 101, return these players as player and the property of the follow edge as degree. These players are grouped and
the sum of their degree values is returned.
-- The following result is returned:
==================
| SUM($-.degree) |
==================
| 186            |
------------------
```

Last update: April 8, 2021

2.3.5 LIMIT Syntax

`LIMIT` works the same as in `SQL`, and must be used with pipe `|`. The `LIMIT` clause accepts one or two arguments. The values of both arguments must be zero or positive integers.

```
ORDER BY <expressions> [ASC | DESC]
LIMIT [<offset_value>,] <number_rows>
```

- **expressions**

  The columns or calculations that you wish to sort.

- **number_rows**

  It constrains the number of rows to return. For example, LIMIT 10 would return the first 10 rows. This is where sorting order matters so be sure to use an `ORDER BY` clause appropriately.

- **offset_value**

  Optional. It defines from which row to start including the rows in the output. The offset starts from zero.

When using `LIMIT`, it is important to use an `ORDER BY` clause that constrains the output into a unique order. Otherwise, you will get an unpredictable subset of the output.

For example:

```
nebula> GO FROM 200 OVER serve REVERSELY YIELD $$.player.name AS Friend, $$.player.age AS Age | ORDER BY Age, Friend | LIMIT 3;
========================
| Friend          | Age |
========================
| Kyle Anderson   | 25  |
------------------------
| Aron Baynes     | 32  |
------------------------
| Marco Belinelli | 32  |
```

Last update: April 8, 2021

2.3.6 Logical Operators

Name	Description
&&	Logical AND
!	Logical NOT
||	Logical OR
XOR	Logical XOR

In nGQL, non-zero numbers are evaluated to *true*. For the precedence of the operators, refer to Operator Precedence.

- &&

Logical AND:

```
nebula> YIELD -1 && true;
==============
| (-1&&true) |
==============
| true       |
--------------
```

- !

Logical NOT:

```
nebula> YIELD !(-1);
=========
| !(-1) |
=========
| false |
---------
```

- ||

Logical OR:

```
nebula> YIELD 1 || !1;
=============
| (1||!(1)) |
=============
| true      |
-------------
```

- ^

Logical XOR:

```
nebula> YIELD (NOT 0 || 0) AND 0 XOR 1 AS ret;
=========
| ret   |
=========
| true  |
---------
```

Last update: April 8, 2021

2.3.7 Operator Precedence

The following list shows the precedence of nGQL operators in descending order. Operators that are shown together on a line have the same precedence.

```
- (negative number)
!
*, /, %
-, +
== , >=, >, <=, <, <>, !=
&&
||
= (assignment)
```

For operators that occur at the same precedence level within an expression, evaluation proceeds left to right, with the exception that assignments evaluate right to left.

The precedence of operators determines the order of evaluation of terms in an expression. To override this order and group terms explicitly, use parentheses.

Examples:

```
nebula> YIELD 2+3*5;
nebula> YIELD (2+3)*5;
```

Last update: April 8, 2021

2.3.8 ORDER BY Function

Similar with SQL, `ORDER BY` can be used to sort in ascending (`ASC`) or descending (`DESC`) order for returned results. `ORDER BY` can only be used in the `PIPE` -syntax (`|`).

```
ORDER BY <expression> [ASC | DESC] [, <expression> [ASC | DESC] ...]
```

By default, `ORDER BY` sorts the records in ascending order if no `ASC` or `DESC` is given.

Example

```
nebula> FETCH PROP ON player 100,101,102,103 YIELD player.age AS age, player.name AS name | ORDER BY age, name DESC;

-- Fetch four vertices and sort them by their ages in ascending order, and for those in the same age, sort them by name in descending order.
-- The following result is returned:
====================================
| VertexID | age | name              |
====================================
| 103      | 32  | Rudy Gay          |
------------------------------------
| 102      | 33  | LaMarcus Aldridge |
------------------------------------
| 101      | 36  | Tony Parker       |
------------------------------------
| 100      | 42  | Tim Duncan        |
------------------------------------
```

(See FETCH for the usage.)

```
nebula> GO FROM 100 OVER follow YIELD $$.player.age AS age, $$.player.name AS name | ORDER BY age DESC, name ASC;

-- Search all the players followed by vertex 100 and return their ages and names. The age is in descending order; the name is in ascending order if they have
the same name.
-- The following result is returned:
==========================
| age | name              |
==========================
| 36  | Tony Parker       |
--------------------------
| 33  | LaMarcus Aldridge |
--------------------------
| 25  | Kyle Anderson     |
--------------------------
```

Last update: April 8, 2021

2.3.9 Set Operations (`UNION`, `INTERSECT`, and `MINUS`)

You can combine multiple queries using the set operators `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`. All set operators have equal precedence. If a nGQL statement contains multiple set operators, then **Nebula Graph** evaluates them from the left to right unless parentheses explicitly specify another order.

The return results in the `GO` lists of a compound query must match in number and must be in the same datatype group (such as numeric or character).

UNION, UNION DISTINCT, and UNION ALL

Operator `UNION DISTINCT` (or by short `UNION`) returns the union of two sets A and B (denoted by `A ⋃ B` in mathematics), with the distinct element belongs to set A or set B, or both.

Meanwhile, operation `UNION ALL` returns the union set with duplicated elements. The `UNION` syntax is

```
<left> UNION [DISTINCT | ALL] <right> [ UNION [DISTINCT | ALL] <right> ...]
```

where `<left>` and `<right>` must have the same number of columns and pair-wise data types. If the data types are different, **Nebula Graph** will convert according to Type Conversion.

EXAMPLE

The following statement

```
nebula> GO FROM 1 OVER e1 \
        UNION \
        GO FROM 2 OVER e1
```

returns the neighbors' id of vertex `1` and `2` (along with edge `e1`) without duplication.

While

```
nebula> GO FROM 1 OVER e1 \
        UNION ALL\
        GO FROM 2 OVER e1
```

returns all the neighbors of vertex `1` and `2`, with all possible duplications.

`UNION` can also work with the `YIELD` statement. For example, let's suppose the results of the following two queries.

```
nebula> GO FROM 1 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2; -- query 1
======================
| id  | col_1| col_2  |
======================
| 104 |   1  |    2   |    -- line 1
----------------------
| 215 |   4  |    3   |    -- line 3
----------------------

nebula> GO FROM 2,3 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2;  -- query 2
======================
| id  | col_1| col_2 |
======================
| 104 |   1  |    2  |    -- line 1
----------------------
| 104 |   2  |    2  |    -- line 2
----------------------
```

And the following statement

```
nebula> GO FROM 1 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2   \
        UNION /* DISTINCT */     \
        GO FROM 2,3 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2;
```

will return as follows:

```
======================
| id  | col_1| col_2  |    -- UNION or UNION DISTINCT. The column names come from query 1
======================
```

```
| 104 |  1   |    2   |    -- line 1
----------------------
| 104 |  2   |    2   |    -- line 2
----------------------
| 215 |  4   |    3   |    -- line 3
----------------------
```

Notice that line 1 and line 2 return the same id (104) with different column values. The `DISTINCT` check duplication by all the columns for every line. So line 1 and line 2 are different.

You can expect the `UNION ALL` result

```
nebula> GO FROM 1 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2   \
        UNION ALL   \
        GO FROM 2,3 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2;

======================
| id  | col_1| col_2 |    -- UNION ALL
======================
| 104 |   1  |   2   |    -- line 1
----------------------
| 104 |   1  |   2   |    -- line 1
----------------------
| 104 |   2  |   2   |    -- line 2
----------------------
| 215 |   4  |   3   |    -- line 3
----------------------
```

INTERSECT

Operator `INTERSECT` returns the intersection of two sets A and B (denoted by A ⋂ B), if the elements belong both to set A and set B.

```
<left> INTERSECT <right>
```

Alike `UNION`, `<left>` and `<right>` must have the same number of columns and data types. Besides, only the same line of `<left>` and `<right>` will be returned.

For example, the following query

```
nebula> GO FROM 1 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2
INTERSECT
GO FROM 2,3 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2;
```

returns

```
======================
| id  | col_1 | col_2 |
======================
| 104 |   1   |    2  |    -- line 1
----------------------
```

MINUS

The set subtraction (or difference), A - B, consists of elements that are in A but not in B. So the operation order matters.

For example, the following query

```
nebula> GO FROM 1 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2
MINUS
GO FROM 2,3 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2;
```

comes out

```
========================
| id  | col_1 | col_2  |
========================
| 215 |   4   |    3   |     -- line 3
------------------------
```

And if we reverse the `MINUS` order, the query

```
nebula> GO FROM 2,3 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2
MINUS
GO FROM 1 OVER e1 YIELD e1._dst AS id, e1.prop1 AS col_1, $$.tag.prop2 AS col_2;
```

returns

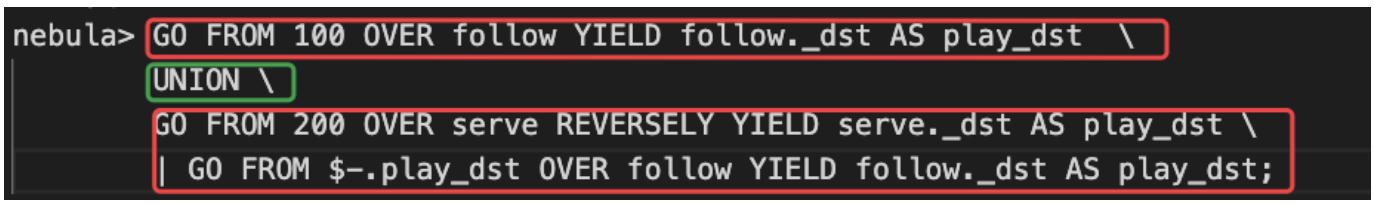
```
=======================
| id  | col_1 | col_2 |    -- column named from query 2
=======================
| 104 |    2  |    2  |    -- line 2
-----------------------
```

Precedence of the SET Operations and Pipe

Please note that when a query contains pipe `|` and set operations, pipe takes precedence. Refer to the Pipe Doc for details. Query `GO FROM 1 UNION GO FROM 2 | GO FROM 3` is the same as query `GO FROM 1 UNION (GO FROM 2 | GO FROM 3)`.

For example:

```
nebula> GO FROM 100 OVER follow YIELD follow._dst AS play_dst  \
        UNION \
        GO FROM 200 OVER serve REVERSELY YIELD serve._dst AS play_dst \
        | GO FROM $-.play_dst OVER follow YIELD follow._dst AS play_dst;
```



The statements in the red bar are executed first concurrently, and then the statement in the green box is executed.

```
nebula> (GO FROM 100 OVER follow YIELD follow._dst AS play_dst  \
        UNION \
        GO FROM 200 OVER serve REVERSELY YIELD serve._dst AS play_dst) \
        | GO FROM $-.play_dst OVER follow YIELD follow._dst AS play_dst;
```

In the above query, the parentheses change the execution priority, and the statements within the parentheses take the precedence.

Last update: April 8, 2021

2.3.10 String Comparison Functions and Operators

Name	Description
CONTAINS	Perform case-sensitive inclusion searching in strings

- CONTAINS

The `CONTAINS` operator is used to perform case-sensitive matching regardless of location within a string. The `CONTAINS` operator requires string type in both left and right side.

```
nebula> GO FROM 107 OVER serve WHERE $$.team.name CONTAINS "riors" \
        YIELD $^.player.name, serve.start_year, serve.end_year, $$.team.name;
==========================================================================
| $^.player.name | serve.start_year | serve.end_year | $$.team.name |
==========================================================================
| Aron Baynes    | 2001             | 2009           | Warriors     |
--------------------------------------------------------------------------

nebula> GO FROM 107 OVER serve WHERE $$.team.name CONTAINS "Riors" \
        YIELD $^.player.name, serve.start_year, serve.end_year, $$.team.name; -- The follow query returns nothing.
```

```
nebula> GO FROM 107 OVER serve WHERE (STRING)serve.start_year CONTAINS "07" && \
        $^.player.name CONTAINS "Aron" \
        YIELD $^.player.name, serve.start_year, serve.end_year, $$.team.name;
==========================================================================
| $^.player.name | serve.start_year | serve.end_year | $$.team.name |
==========================================================================
| Aron Baynes    | 2007             | 2010           | Nuggets      |
--------------------------------------------------------------------------
```

```
nebula> GO FROM 107 OVER serve WHERE !($$.team.name CONTAINS "riors") \
        YIELD $^.player.name, serve.start_year, serve.end_year, $$.team.name;
==========================================================================
| $^.player.name | serve.start_year | serve.end_year | $$.team.name |
==========================================================================
| Aron Baynes    | 2007             | 2010           | Nuggets      |
--------------------------------------------------------------------------
```

Last update: April 8, 2021

2.3.11 UUID

`UUID` is used to generate the global unique identifiers.

When the number of vertices reaches billions, using `Hash` Function to generate vids has a certain conflict probability. Therefore, **Nebula Graph** provides `UUID` Function to avoid vid conflicts in a large number of vertices. `UUID` Function is composed of the `Murmur` hash function and the current timestamp (in seconds).

Low Performance and compatibility issues for UUID

Values generated with the `UUID` are stored in the **Nebula Graph** Storage service in key-value mode. When called, it checks whether this key exists or conflicts. So the performance may be lower than hash. In addition, `UUID` may not be available in future versions.

Insert with `UUID`:

```
-- Insert a vertex with the UUID function.
nebula> INSERT VERTEX player (name, age) VALUES uuid("n0"):("n0", 21);
-- Insert an edge with the UUID function.
nebula> INSERT EDGE follow(degree) VALUES uuid("n0") -> uuid("n1"): (90);
```

Fetch with `UUID`:

```
nebula> FETCH PROP ON player uuid("n0") YIELD player.name, player.age;
-- The following result is returned:
================================================
| VertexID             | player.name | player.age |
================================================
| -5057115778034027261 | n0          | 21         |
------------------------------------------------
nebula> FETCH PROP ON follow uuid("n0") -> uuid("n1");
-- The following result is returned:
================================================================================
| follow._src          | follow._dst         | follow._rank | follow.degree |
================================================================================
| -5057115778034027261 | 4039977434270020867 | 0            | 90            |
--------------------------------------------------------------------------------
```

`Go` with `UUID`:

```
nebula> GO FROM uuid("n0") OVER follow;
--The following result is returned:
======================
| follow._dst         |
======================
| 4039977434270020867 |
----------------------
```

Last update: April 8, 2021

2.4 Language Structure

2.4.1 Literal Values

Boolean Literals

The boolean literals `TRUE` and `FALSE` can be written in any letter case.

```
nebula> yield TRUE, true, FALSE, false, FalsE
=========================================
| true  | true  | false | false | false |
=========================================
| true  | true  | false | false | false |
-----------------------------------------
```

Last update: April 8, 2021

Numeric Literals

Numeric literals include integers literals and floating-point literals (doubles).

INTEGERS LITERALS

Integers are 64 bit digitals, and can be preceded by + or - to indicate a positive or negative value, respectively. They're the same as `int64_t` in the C language.

Notice that the maximum value for the positive integers is `9223372036854775807`. It's syntax-error if you try to input any value greater than the maximum. So as the minimum value `-9223372036854775808` for the negative integers.

FLOATING-POINT LITERALS (DOUBLES)

Floating-points are the same as `double` in the `C` language.

The range for double is about `-1.79769e+308` to `1.79769e+308`.

Scientific notations is not supported yet.

Scientific Notations

Nebula Graph supports using scientific notation to represent the Double type. For example:

```
nebula> CREATE TAG test1(price DOUBLE);
nebula> INSERT VERTEX test1(price) VALUES 100:(1.2E3);
```

Last update: April 8, 2021

String Literals

A string is a sequence of bytes or characters, enclosed within either single quote (') or double quote (") characters. Examples:

```
nebula> YIELD 'a string'
nebula> YIELD "another string"
```

Certain backslash escapes (\) are supported (also known as the *escape characters*). They are shown in the following table:

Escape Sequence	Character Represented by Sequence
\'	A single quote (') character
\"	A double quote (") character
\t	A tab character
\n	A newline character
\b	A backspace character
\	A backslash (\) character

Here are some examples:

```
nebula> YIELD 'This\nIs\nFour\nLines'
========================
| "This
Is
Four
Lines" |
========================
| This
Is
Four
Lines   |
------------------------

nebula> YIELD 'disappearing\ backslash'
============================
| "disappearing backslash" |
============================
| disappearing backslash   |
----------------------------
```

Last update: April 8, 2021

2.4.2 Comment Syntax

Nebula Graph supports four comment styles:

- From a # character to the end of the line.
- From a -- sequence to the end of the line. When using '--' as a comment, you need to add a space after it, i.e. '-- '.
- From a // sequence to the end of the line, as in the C programming language.
- From a `/*  */` sequence. This syntax enables a comment to extend over multiple lines because the beginning and closing sequences need not be on the same line.

Nested comments are not supported. The following example demonstrates all these comment styles:

```
nebula> -- Do nothing this line
nebula> YIELD 1+1;     # This comment continues to the end of line
nebula> YIELD 1+1;     -- This comment continues to the end of line
nebula> YIELD 1+1;     // This comment continues to the end of line
nebula> YIELD 1      /* This is an in-line comment */ + 1;
nebula> YIELD 11 +             \
/* Multiple-line comment       \
Use backslash as line break.   \
*/ 12;
```

The backslash `\` in a line indicates a line break.

Last update: April 8, 2021

2.4.3 Identifer Case Sensitivity

Identifiers are Case-Sensitive

The following statements would not work because they refer to two different spaces, i.e. `my_space` and `MY_SPACE`:

```
nebula> CREATE SPACE my_space;
nebula> use MY_SPACE;  -- my_space and MY_SPACE are two different spaces
```

Keywords and Reserved Words are Case-Insensitive

The following statements are equivalent:

```
nebula> show spaces;  -- show and spaces are keywords.
nebula> SHOW SPACES;
nebula> SHOW spaces;
nebula> show SPACES;
```

Last update: April 8, 2021

2.4.4 Keywords and Reserved Words

Keywords are words that have significance in nGQL. Certain keywords are reserved and require special treatment for use as identifiers.

Non-reserved keywords are permitted as identifiers without quoting. All the non-reserved keywords are automatically converted to lower case. Non-reserved keywords are case-insensitive. Reserved words are permitted as identifiers if you quote them with back quotes such as `AND`.

```
nebula> CREATE TAG TAG(name string);
[ERROR (-7)]: SyntaxError: syntax error near `TAG'

nebula> CREATE TAG SPACE(name string); -- SPACE is an unreserved KEY WORD
Execution succeeded

nebula> SHOW TAGS; -- All the non-reserved keywords are automatically converted to lower case.
=============
| ID | Name |
=============
| 25 | space|
-------------
```

`TAG` is a reserved keyword and must be quoted with backtick to be used as an identifier. `SPACE` is keyword but not reserved, so its use as identifiers does not require quoting.

```
nebula> CREATE TAG `TAG` (name string); -- TAG is a reserved word here
Execution succeeded
```

Reserved Words

The following list shows reserved words in nGQL.

```
ADD
ALTER
AND
AS
ASC
BALANCE
BIGINT
BOOL
BY
CHANGE
COMPACT
CREATE
DELETE
DESC
DESCRIBE
DISTINCT
DOUBLE
DOWNLOAD
DROP
EDGE
EDGES
EXISTS
FETCH
FIND
FLUSH
FROM
GET
GO
GRANT
IF
IN
INDEX
INDEXES
INGEST
INSERT
INT
INTERSECT
IS
LIMIT
LOOKUP
MATCH
MINUS
NO
NOT
NULL
OF
OFFSET
ON
OR
```

```
ORDER
OVER
OVERWRITE
PROP
REBUILD
RECOVER
REMOVE
RETURN
REVERSELY
REVOKE
SET
SHOW
STEPS
STOP
STRING
SUBMIT
TAG
TAGS
TIMESTAMP
TO
UNION
UPDATE
UPSERT
UPTO
USE
VERTEX
WHEN
WHERE
WITH
XOR
YIELD
```

Non-Reserved Keywords

```
ACCOUNT
ADMIN
ALL
AVG
BIDIRECT
BIT_AND
BIT_OR
BIT_XOR
CHARSET
COLLATE
COLLATION
CONFIGS
COUNT
COUNT_DISTINCT
DATA
DBA
DEFAULT
FORCE
GOD
GRAPH
GROUP
GUEST
HDFS
HOSTS
JOB
JOBS
LEADER
MAX
META
MIN
OFFLINE
PART
PARTITION_NUM
PARTS
PASSWORD
PATH
REPLICA_FACTOR
ROLE
ROLES
SHORTEST
SNAPSHOT
SNAPSHOTS
SPACE
SPACES
STATUS
STD
STORAGE
SUM
TTL_COL
TTL_DURATION
USER
USERS
UUID
VALUES
```

Last update: April 8, 2021

2.4.5 PIPE Syntax

One major difference between nGQL and SQL is how sub-queries are composed.

In SQL, sub-queries are nested (embedded) to form a statement. Meanwhile, nGQL uses shell style `PIPE (|)`.

Examples

```
nebula> GO FROM 100 OVER follow YIELD follow._dst AS dstid, $$.player.name AS Name | \
GO FROM $-.dstid OVER follow YIELD follow._dst, follow.degree, $-.Name
```

The destination (vertex) `id` will be given as the default value if no `YIELD` is used.

But if `YIELD` is declared explicitly, (the default value) `id` will not be given.

The alias name mentioned right after placeholder `$-.` must be defined in the previous `YIELD` statement, such as `dstid` or `Name` as shown in the above example.

Last update: April 8, 2021

2.4.6 Property Reference

You can refer a vertex or edge's property in `WHERE` or `YIELD` syntax.

Reference From Vertex

FOR SOURCE VERTEX

```
$^.tag_name.prop_name
```

where symbol `$^` is used to get a source vertex's property, `tag_name` indicates the source vertex's `tag`, and `prop_name` specifies the property name.

FOR DESTINATION VERTEX

```
$$.tag_name.prop_name
```

Symbol `$$` indicates the ending vertex, `tag_name` and `prop_name` are the vertex's tag and property respectively.

EXAMPLE

```
nebula> GO FROM 100 OVER follow YIELD $^.player.name AS startName, $$.player.age AS endAge;
```

Use the above query to get the source vertex's property name and ending vertex's property age.

Reference From Edge

FOR PROPERTY

You can use the following syntax to get an edge's property.

```
edge_type.edge_prop
```

`edge_type` is the edge's type, meanwhile `edge_prop` is the property.

For example,

```
nebula> GO FROM 100 OVER follow YIELD follow.degree;
```

FOR BUILT-IN PROPERTIES

There are four built-in properties in the edge:

- _src: source vertex ID of the edge
- _dst: destination ID of the edge
- _type: edge type
- _rank: the edge's rank

You can use `_src` and `_dst` to get the starting and ending vertices' ID, and they are very commonly used to show a graph path.

For example,

```
nebula> GO FROM 100 OVER follow YIELD follow._src, follow._dst, follow._type, follow._rank;
=============================================================
| follow._src | follow._dst | follow._type | follow._rank |
=============================================================
| 100         | 101         | 26           | 0            |
-------------------------------------------------------------
| 100         | 102         | 26           | 0            |
-------------------------------------------------------------
| 100         | 106         | 26           | 0            |
-------------------------------------------------------------
```

This statement returns all the neighbors of vertex `100` over edge type `follow`, by referencing `follow._src` as the starting vertex ID (which, of course, is `100`) and `follow._dst` as the ending vertex ID.

Last update: April 8, 2021

2.4.7 Schema Object Names

Certain objects within **Nebula graph**, including space, tag, edge, alias, customer variables and other object names are referred as identifiers. This section describes the rules for identifiers in **Nebula Graph**:

- Permitted characters in identifiers: ASCII: [0-9,a-z,A-Z,_] (basic Latin letters, digits 0-9, underscore), other punctuation characters are not supported.
- All identifiers must begin with a letter of the alphabet.
- Identifiers are case sensitive.
- You cannot use a keyword (a reserved word) as an identifier.

Last update: April 8, 2021

2.4.8 Statement Composition

There are only two ways to compose statements (or sub-queries):

- More than one statements can be batched together, separated by semicolon (;). The result of the last statement will be returned as the result of the batch.
- Statements could be piped together using operator (|), much like the pipe in the shell scripts. The result yielded from the previous statement could be redirected to the next statement as input.

Notice that compose statements are not `Transactional` queries. For example, a statement composed of three sub-queries: A | B | C, where A is a read operation, B is a computation, and C is a write operation. If any part fails in the execution, the whole result could be undefined -- currently, there is no so called roll back -- what was written depends on the query executor.

Examples

- semicolon statements

```
SHOW TAGS; SHOW EDGES;          -- only edges are shown

INSERT VERTEX player(name, age) VALUES 100:("Tim Duncan", 42); \
INSERT VERTEX player(name, age) VALUES 101:("Tony Parker", 36); \
INSERT VERTEX player(name, age) VALUES 102:("LaMarcus Aldridge", 33);  /* multiple vertices are added in a compose statement. */
```

- PIPE statements

```
GO FROM 201 OVER edge_serve | GO FROM $-.id OVER edge_fans | GO FROM $-.id ...
```

Placeholder `$-.id` takes the result from the first statement `GO FROM 201 OVER edge_serve YIELD edge_serve._dst AS id`.

Last update: April 8, 2021

2.4.9 User-Defined Variables

Nebula Graph supports user-defined variables, which allows passing the result of one statement to another. A user-defined variable is written as `$var_name`, where `var_name` is a user-defined name/variable that consists of alphanumeric characters, any other characters are not recommended currently.

User-defined variables can only be used in one execution (compound statements separated by semicolon `;` or pipe `|` and are submitted to the server to execute together).

Be noted that a user-defined variable is valid only at the current session and execution. A user-defined variable in one statement can NOT be used in neither other clients nor other executions, which means that the definition statement and the statements that use it must be submitted together. And when the session ends these variables are automatically expired.

User-defined variables are case-sensitive.

Example:

```
nebula> $var = GO FROM hash('curry') OVER follow YIELD follow._dst AS id; \
GO FROM $var.id OVER serve;
```

Last update: April 8, 2021

2.5 Statement Syntax

2.5.1 Data Definition Statements

ALTER EDGE Syntax

```
ALTER EDGE <edge_name>
    <alter_definition> [, alter_definition] ...]
    [ttl_definition [, ttl_definition] ... ]

alter_definition:
| ADD    (prop_name data_type)
| DROP   (prop_name)
| CHANGE (prop_name data_type)

ttl_definition:
    TTL_DURATION = ttl_duration, TTL_COL = prop_name
```

`ALTER EDGE` statement changes the structure of an edge. For example, you can add or delete properties, change the data type of an existing property. You can also set a property as TTL (Time-To-Live), or change the TTL duration.

> **NOTE**: **Nebula Graph** automatically examines indexes when altering an edge. When altering an edge, **Nebula Graph** first checks whether the edge is associated with any indexes then traverses all of them to check whether the column item to be dropped or changed exists in the index column. If existed, the alter is rejected. Otherwise, it is allowed.

Please refer to Index Documentation on details about index.

Multiple `ADD`, `DROP`, and `CHANGE` clauses are permitted in a single `ALTER` statements, separated by commas. But do NOT add, drop, change the same property in one statement. If you have to do so, make each operation as a clause of the `ALTER` statement.

```
nebula> CREATE EDGE e1 (prop3 int, prop4 int, prop5 int);
nebula> ALTER EDGE e1 ADD (prop1 int, prop2 string),    /* ▒▒ prop1 */
              CHANGE (prop3 string),            /* ▒ prop3 ▒▒▒▒▒▒▒ */
              DROP (prop4, prop5);               /* ▒▒ prop4 ▒ prop5 */

nebula> ALTER EDGE e1 TTL_DURATION = 2, TTL_COL = "prop1";
```

> **NOTE**: `TTL_COL` only supports the properties whose values are of the `INT` or the `TIMESTAMP` type.

Last update: April 8, 2021

ALTER TAG Syntax

```
ALTER TAG <tag_name>
    <alter_definition> [, alter_definition] ...]
    [ttl_definition [, ttl_definition] ... ]

alter_definition:
| ADD    (prop_name data_type)
| DROP   (prop_name)
| CHANGE (prop_name data_type)

ttl_definition:
    TTL_DURATION = ttl_duration, TTL_COL = prop_name
```

`ALTER TAG` statement changes the structure of a tag. For example, you can add or delete properties, change the data type of an existing property. You can also set a property as TTL (Time-To-Live), or change the TTL duration.

> **NOTE**: **Nebula Graph** automatically examines indexes when altering a tag. When altering a tag, **Nebula Graph** first checks whether the tag is associated with any indexes then traverses all of them to check whether the column item to be dropped or changed exists in the index column. If existed, the alter is rejected. Otherwise, it is allowed.

Please refer to Index Documentation on details about index.

Multiple `ADD`, `DROP`, and `CHANGE` clauses are permitted in a single `ALTER` statements, separated by commas. But do NOT add, drop, change the same property in one statement. If you have to do so, make each operation as a clause of the `ALTER` statement.

```
nebula> CREATE TAG t1 (name string, age int);
nebula> ALTER TAG t1 ADD (id int, address string);
nebula> ALTER TAG t1 TTL_DURATION = 2, TTL_COL = "age";
```

> **NOTE**: `TTL_COL` only supports the properties whose values are of the `INT` or the `TIMESTAMP` type.

Last update: April 8, 2021

CREATE SPACE Syntax

```
CREATE SPACE [IF NOT EXISTS] <space_name>
   [(partition_num = <part_num>, replica_factor = <raft_copy>, charset = <charset>, collate = <collate>)]
```

This statement creates a new space with the given name. SPACE is a region that provides physically isolated graphs in **Nebula Graph**. An error occurs if the database exists.

IF NOT EXISTS

You can use the `If NOT EXISTS` keywords when creating spaces. This keyword automatically detects if the corresponding space exists. If it does not exist, a new one is created. Otherwise, no space is created.

NOTE: The space existence detection here only compares the space name (excluding properties).

SPACE NAME

- **space_name**

  The name uniquely identifies the space in a cluster. The rules for the naming are given in Schema Object Names

CUSTOMIZED SPACE OPTIONS

When creating a space, the following four customized options can be given:

- *partition_num*

  partition_num specifies the number of partitions in one replica. The default value is 100. It is usually 5 times the number of hard disks in the cluster.

- *replica_factor*

  replica_factor specifies the number of replicas in the cluster. The default replica factor is 1. The suggested number is 3 in cluster. It is usually 3 in production. Due to the majority voting principle, it must set to be odd.

- *charset*

  charset is short for character set. A character set is a set of symbols and encodings. The default value is utf8.

- *collate*

  A *collation* is a set of rules for comparing characters in a character set. The default value is utf8_bin.

However, if no option is given, **Nebula Graph** will create the space with the default partition number, replica factor, charset and collate.

EXAMPLE

```
nebula> CREATE SPACE my_space_1; -- create space with default partition number and replica factor
nebula> CREATE SPACE my_space_2(partition_num=10); -- create space with default replica factor
nebula> CREATE SPACE my_space_3(replica_factor=1); -- create space with default partition number
nebula> CREATE SPACE my_space_4(partition_num=10, replica_factor=1);
```

CHECKING PARTITION DISTRIBUTION

On some large clusters, due to the different startup time, the partition distribution may be unbalanced. You can check the machine and distribution by the following command (SHOW HOSTS).

```
nebula> SHOW HOSTS;
=============================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=============================================================================================
| 192.168.8.210 | 34600 | online | 13           | test: 13            | test: 37               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34900 | online | 12           | test: 12            | test: 38               |
---------------------------------------------------------------------------------------------
```

If all the machines are online status, but the partition distribution is unbalanced, you can use the following command (BALANCE LEADER) to redistribute the partitions.

```
nebula> BALANCE LEADER;
```

Details see SHOW HOSTS and BALANCE.

Last update: April 8, 2021

CREATE EDGE Syntax

```
CREATE EDGE [IF NOT EXISTS] <edge_name>
    ([<create_definition>, ...])
    [edge_options]

<create_definition> ::=
    <prop_name> <data_type>

<edge_options> ::=
    <option> [, <option> ...]

<option> ::=
    TTL_DURATION [=] <ttl_duration>
    | TTL_COL [=] <prop_name>
    | DEFAULT <default_value>
```

The schema for Nebula Graph is composed of tags and edges, either of which can have properties. The `CREATE EDGE` statement defines an edge type with the given name.

The features of this syntax are described in the following sections:

IF NOT EXISTS

You can use the `If NOT EXISTS` keywords when creating edge types. This keyword automatically detects if the corresponding edge type exists. If it does not exist, a new one is created. Otherwise, no edge type is created.

NOTE: The edge type existence detection here only compares the edge edge name (excluding properties).

EDGE TYPE NAME

- **edge_name**

  The name of edge types must be **unique** within the space. Once the name is defined, it can not be altered. The rules of edge type names are the same as those for names of spaces. See Schema Object Name for detail.

Property Name and Data Type

- **prop_name**

  prop_name indicates the name of properties. It must be unique for each edge type.

- **data_type**

  data_type represents the data type of each property. For more information about data types that Nebula Graph supports, see the data-type section.

  NULL and NOT NULL constrain are not supported yet when creating edge types (comparing with relational databases).

- **Default Constraint**

  You can set the default value of a property when creating an edge type with the `DEFAULT` constraint. The default value will be added to all new edges if no value is specified. The default value can be any of the data type supported by Nebula Graph or expressions. Also, you can specify a value if you don't want to use the default one.

  Using `Alter` to change the default value is not supported.

Time-to-Live (TTL) Syntax

- TTL_DURATION

  ttl_duration specifies the life cycle of vertices (or edges). Data that exceeds the specified TTL will expire. The expiration threshold is the specified TTL_COL value plus the TTL_DURATION.

  If the value for ttl_duration is zero, the vertices or edges will not expire.

- TTL_COL

  The data type of prop_name must be either int64 or timestamp.

- single TTL definition

  Only a single TTL_COL field can be specified.

Details about TTL refer to the TTL Doc.

Examples

```
nebula> CREATE EDGE follow(start_time timestamp, grade double);
nebula> CREATE EDGE noedge();  -- empty properties

nebula> CREATE EDGE follow_with_default(start_time timestamp DEFAULT 0, grade double DEFAULT 0.0);  -- start_time is set to 0 by default, grade is set to 0.0
by default
```

```
nebula> CREATE EDGE marriage(location string, since timestamp)
    TTL_DURATION = 0, TTL_COL = "since"; -- zero, will not expire
```

Last update: April 8, 2021

CREATE TAG Syntax

```
CREATE TAG [IF NOT EXISTS] <tag_name>
    ([<create_definition>, ...])
    [tag_options]

<create_definition> ::=
    <prop_name> <data_type>

<tag_options> ::=
    <option> [, <option> ...]

<option> ::=
    TTL_DURATION [=] <ttl_duration>
    | TTL_COL [=] <prop_name>
    | DEFAULT <default_value>
```

Nebula Graph's schema is composed of tags and edges, either of which may have properties. `CREATE TAG` statement defines a tag with the given name.

The features of this syntax are described in the following sections:

IF NOT EXISTS

You can use the `If NOT EXISTS` keywords when creating tags. This keyword automatically detects if the corresponding tag exists. If it does not exist, a new one is created. Otherwise, no tag is created.

NOTE: The tag existence detection here only compares the tag name (excluding properties).

TAG NAME

- **tag_name**

  The name of tags must be **unique** within the space. Once the name is defined, it can not be altered. The rules of tag names are the same as those for names of spaces. See Schema Object Name for detail.

Property Name and Data Type

- **prop_name**

  prop_name indicates the name of properties. It must be unique for each tag.

- **data_type**

  data_type represents the data type of each property. For more information about data types that **Nebula Graph** supports, see data-type section.

  NULL and NOT NULL constrain are not supported yet when creating tags (comparing with relational databases).

- **Default Constraint**

  You can set the default value of a property when creating a tag with the `DEFAULT` constraint. The default value will be added to all new vertices if no other value is specified. The default value can be any of the data type supported by **Nebula Graph** or expressions. Also you can write a user-specified value if you don't want to use the default one.

  Using `Alter` to change the default value is not supported.

Time-to-Live (TTL) Syntax

- TTL_DURATION

  ttl_duration specifies the life cycle of vertices (or edges). Data that exceeds the specified TTL will expire. The expiration threshold is the specified TTL_COL value plus the TTL_DURATION.

  If the value for ttl_duration is zero, the vertices or edges will not expire.

- TTL_COL

  The data type of prop_name must be either int64 or timestamp.

- single TTL definition

  Only a single TTL_COL field can be specified.

Details about TTL refer to the TTL Doc.

Examples

```
nebula> CREATE TAG course(name string, credits int);
nebula> CREATE TAG notag();  -- empty properties

nebula> CREATE TAG player_with_default(name string, age int DEFAULT 20);  -- age is set to 20 by default
```

```
nebula> CREATE TAG woman(name string, age int,
   married bool, salary double, create_time timestamp)
   TTL_DURATION = 100, TTL_COL = "create_time"; -- time interval is 100s, starting from the create_time filed

nebula> CREATE TAG icecream(made timestamp, temperature int)
   TTL_DURATION = 100, TTL_COL = "made",
   --  Data expires after TTL_DURATION
```

Last update: April 8, 2021

DROP EDGE Syntax

```
DROP EDGE [IF EXISTS] <edge_type_name>
```

You must have the DROP privilege for the edge type.

> **NOTE**: When dropping an edge, **Nebula Graph** only checks whether the edge is associated with any indexes. If so the deletion is rejected.

Please refer to Index Documentation on details about index.

You can use the `If EXISTS` keywords when dropping edges. These keywords automatically detect if the corresponding edge exists. If it exists, it will be deleted. Otherwise, no edge is deleted.

This statement removes all the edges (connections) within the specific edge type.

This operation only deletes the Schema data, all the files and directories in the disk are NOT deleted directly, data is deleted in the next compaction.

Last update: April 8, 2021

DROP TAG Syntax

```
DROP TAG [IF EXISTS] <tag_name>
```

You must have the DROP privilege for the tag.

> **NOTE**: Be careful with this statement. When dropping a tag, **Nebula Graph** will only check whether the tag is associated with any indexes. If so the deletion is rejected.

Please refer to Index Documentation on details about index.

You can use the `If EXISTS` keywords when dropping tags. These keywords automatically detect if the corresponding tag exists. If it exists, it will be deleted. Otherwise, no tag is deleted.

A vertex can have one or more tags (types).

If a vertex has only one tag, after the tag is dropped, the vertex can NOT be accessible. But its edges are available. If a vertex has multiple tags, after one tag is dropped, the vertex is still accessible. But all the properties defined by this dropped tag are not accessible.

This operation only deletes the Schema data, all the files and directories in the disk are NOT deleted directly, data is deleted in the next compaction.

Last update: April 8, 2021

DROP SPACE Syntax

```
DROP SPACE [IF EXISTS] <space_name>
```

You must have the DROP privilege for the graph space.

DROP SPACE deletes everything in the specific space.

You can use the `If EXISTS` keywords when dropping spaces. This keyword automatically detects if the corresponding space exists. If it exists, it will be deleted. Otherwise, no space is deleted.

Other spaces remain unchanged.

This statement does not immediately remove all the files and directories in the storage engine (and release disk space). The deletion depends on the implementation of different storage engines.

> Be *very* careful with this statement.

Last update: April 8, 2021

Schema Index

```
CREATE {TAG | EDGE} INDEX [IF NOT EXISTS] <index_name> ON {<tag_name> | <edge_name>} (prop_name_list)
```

Schema indexes are built to fast process graph queries. **Nebula Graph** supports two different kinds of indexing to speed up query processing: **tag indexes** and **edge type indexes**.

Most graph queries start the traversal from a list of vertices or edges that are identified by their properties. Schema indexes make these global retrieval operations efficient on large graphs.

Normally, you create indexes on a tag/edge-type at the time the tag/edge-type itself is created with `CREATE TAG/EDGE` statement.

CREATE INDEX

`CREATE INDEX` enables you to add indexes to existing tag/edge-type.

> **NOTE**: Creating index will affect the write performance.

We suggest you import data first and then rebuild the index in batch.

Create Single-Property Index

```
nebula> CREATE TAG INDEX player_index_0 on player(name);
```

The above statement creates an index for the *name* property on all vertices carrying the *player* tag.

```
nebula> CREATE EDGE INDEX follow_index_0 on follow(degree);
```

The above statement creates an index for the *degree* property on all edges carrying the *follow* edge type.

Create Composite Index

The schema indexes also support spawning over multiple properties. An index on multiple properties is called a composite index.

> **NOTE**: Index across multiple tags is not yet supported.

Consider the following example:

```
nebula> CREATE TAG INDEX player_index_1 on player(name,age);
```

This statement creates a composite index for the *name* and *age* property on all vertices carrying the *player* tag.

SHOW INDEX

```
SHOW {TAG | EDGE} INDEXES
```

`SHOW INDEXES` returns the defined tag/edg-type index information. For example, list the indexes with the following command:

```
nebula> SHOW TAG INDEXES;
============================
| Index ID | Index Name     |
============================
| 22       | player_index_0 |
----------------------------
| 23       | player_index_1 |
----------------------------

nebula> SHOW EDGE INDEXES;
============================
| Index ID | Index Name     |
============================
| 24       | follow_index_0 |
----------------------------
```

DESCRIBE INDEX

```
DESCRIBE {TAG | EDGE} INDEX <index_name>
```

`DESCRIBE INDEX` is used to obtain information about the index. For example, list the index information with the following command:

```
nebula> DESCRIBE TAG INDEX player_index_0;
==================
| Field | Type   |
==================
| name  | string |
------------------

nebula> DESCRIBE TAG INDEX player_index_1;
==================
| Field | Type   |
==================
| name  | string |
------------------
| age   | int    |
------------------
```

DROP INDEX

```
DROP {TAG | EDGE} INDEX [IF EXISTS] <index_name>
```

`DROP INDEX` drops the index named *index_name* from the tag/edge-type. For example, drop the index *player_index_0* with the following command:

```
nebula> DROP TAG INDEX player_index_0;
```

REBUILD INDEX

```
REBUILD {TAG | EDGE} INDEX <index_name> OFFLINE
```

Create Index section describes how to build indexes to improve query performance. If the index is created before data insertion, there is no need to rebuild the index and this section can be skipped; if data is updated or newly inserted before the index creation, it is necessary to rebuild the indexes in order to make sure that the indexes contain the previously added data. If the current database does not provide any services, use the `OFFLINE` keyword to speed up the rebuilding.

After rebuilding is complete, you can use the `SHOW {TAG | EDGE} INDEX STATUS` command to check if the index is successfully rebuilt. For example:

```
nebula> CREATE TAG person(name string, age int, gender string, email string);
Execution succeeded (Time spent: 10.051/11.397 ms)

nebula> CREATE TAG INDEX single_person_index ON person(name);
Execution succeeded (Time spent: 2.168/3.379 ms)

nebula> REBUILD TAG INDEX single_person_index OFFLINE;
Execution succeeded (Time spent: 2.352/3.568 ms)

nebula> SHOW TAG INDEX STATUS;
========================================
| Name                | Tag Index Status |
========================================
| single_person_index | SUCCEEDED        |
----------------------------------------
```

USING INDEX

After the index is created and data is inserted, you can use the LOOKUP statement to query the data.

There is usually no need to specify which indexes to use in a query, **Nebula Graph** will figure that out by itself.

Last update: April 8, 2021

TTL (time-to-live)

With **TTL**, **Nebula Graph** provides the ability to delete the expired vertices or edges automatically. The system will automatically delete the expired data during the compaction phase. Before compaction, query will filter the expired data.

TTl requires `ttl_col` and `ttl_duration` together. `ttl_col` indicates the TTL column, while `ttl_duration` indicates the duration of the TTL. When the sum of the TTL column and the ttl_duration is less than the current time, we consider the data as expired. The `ttl_col` type is integer or timestamp, and is set in seconds. `ttl_duration` is also set in seconds.

TTL CONFIGURATION

- The `ttl_duration` is set in seconds and ranges from 0 to max(int64). If it is set to 0, the vertex properties of this tag does not expire.
- If TTL is set, when the sum of the `ttl_col` and the `ttl_duration` is less than the current time, we consider the vertex properties of this tag as expired in the specified seconds configured by `ttl_duration` has passed since the `ttl_col` field value.
- When the vertex has multiple tags, the TTL of each tag is processed separately.

SETTING A TTL VALUE

- Setting a TTL value for the existed tag.

```
nebula> CREATE TAG t1(a timestamp);
nebula> ALTER TAG t1 ttl_col = "a", ttl_duration = 5; -- Setting ttl
nebula> INSERT VERTEX t1(a) values 101:(now());
```

The vertex 101 property in tag t1 will expire in 5 seconds since specified by now().

- Or you can set the TTL attribute when creating the tag.

```
nebula> CREATE TAG t2(a int, b int, c string) ttl_duration= 100, ttl_col = "a";
nebula> INSERT VERTEX t2(a, b, c) values 102:(1584441231, 30, "Word");
```

The vertex 102 property in tag t2 will expire in 100 seconds since March 17 2020 at 18:33:51 CST i.e. the timestamp is 1584441231.

- When a vertex has multiple TAGs, the TTL of each TAG is independent from each other.

```
nebula> CREATE TAG t3(a string);
nebula> INSERT VERTEX t1(a),t3(a) values 200:(now(), "hello");
```

The vertex 200 property in tag t1 will expire in 5 seconds.

```
nebula> FETCH PROP ON t1 200;
Execution succeeded (Time spent: 5.945/7.492 ms)

nebula> FETCH PROP ON t3 200;
======================
| VertexID | t3.a    |
======================
| 200      | hello   |
----------------------

nebula> FETCH PROP ON * 200;
======================
| VertexID | t3.a    |
======================
| 200      | hello   |
----------------------
```

DROPPING TTL

If you have set a TTL value for a field and later decide do not want it to ever automatically expire, you can drop the TTL value, set it to an empty string or invalidate it by setting it to 0.

```
nebula> ALTER TAG t1 ttl_col = ""; -- drop ttl attribute;
```

Drop the field a with the ttl attribute:

```
nebula> ALTER TAG t1 DROP (a); -- drop ttl_col
```

Invalidate the TTL:

```
nebula> ALTER TAG t1 ttl_duration = 0; -- keep the ttl but the data never expires
```

TIPS ON TTL

- If a field contains a `ttl_col` field, you can't make any change on the field.

```
nebula> CREATE TAG t1(a int, b int, c string) ttl_duration = 100, ttl_col = "a";
nebula> ALTER TAG t1 CHANGE (a string); -- failed
```

- Note that the a tag or an edge cannot have both the TTL attribute and index at the same time, even if the `ttl_col` column is different from that of the index.

```
nebula> CREATE TAG t1(a int, b int, c string) ttl_duration = 100, ttl_col = "a";
nebula> CREATE TAG INDEX id1 ON t1(a); -- failed
```

```
nebula> CREATE TAG t1(a int, b int, c string) ttl_duration = 100, ttl_col = "a";
nebula> CREATE TAG INDEX id1 ON t1(b); -- failed
```

```
nebula> CREATE TAG t1(a int, b int, c string);
nebula> CREATE TAG INDEX id1 ON t1(a);
nebula> ALTER TAG t1 ttl_col = "a", ttl_duration = 100; -- failed
```

- Adding TTL to an edge is similar to a tag.

Last update: April 8, 2021

2.5.2 Data Query and Manipulation Statements

DELETE EDGE Syntax

The `DELETE EDGE` statement is used to delete edges. Given an edge type, the source vertex and the destination vertex, **Nebula Graph** supports `DELETE` the edge, its associated properties and the edge rank. You can also delete an edge with a certain rank. The syntax is as follows:

```
DELETE EDGE <edge_type> <vid> -> <vid>[@<rank>] [, <vid> -> <vid> ...]
```

For example,

```
nebula> DELETE EDGE follow 100 -> 200;
```

The above query deletes an edge whose source vertex is `100`, destination vertex is `200`, and the edge type is `follow`.

Nebula Graph will find the properties associated with the edge and delete all of them. Atomic operation is not guaranteed during the entire process for now, so please retry when failure occurs.

Last update: April 8, 2021

DELETE VERTEX Syntax

Given a list of vertices IDs, hash IDs or UUIDs, **Nebula Graph** supports `DELETE` the vertices and their associated in and out edges, syntax as the follows:

```
DELETE VERTEX <vid_list>
```

For example,

```
nebula> DELETE VERTEX 121;
```

The above query deletes the vertex whose ID is `121`.

Nebula Graph will find the in and out edges associated with the vertices and delete all of them, then delete information related to the vertices. Atomic operation is not guaranteed during the entire process for now, so please retry when failure occurs.

Last update: April 8, 2021

FETCH Syntax

The `FETCH` syntax is used to get vertex/edge's properties.

FETCH VERTEX PROPERTIES

Use `FETCH PROP ON` to return a (list of) vertex's properties. Currently, you can get multiple vertices' properties with the same tag in one statement. You can use `FETCH` together with pipe and user defined variables.

```
FETCH PROP ON {<tag_name> | <tag_name_list> | *} <vertex_id_list> [YIELD [DISTINCT] <return_list>]
```

Fetch `<tag_name_list>` is only available for version 1.2.1 or later version. If your **Nebula Graph** is earlier than 1.2.1, you can only fetch one type of vertices in a single query.

`*` indicates returning all the properties of the given vertex.

`<tag_name_list>::=[tag_name [, tag_name]]` is the tag name. It must be the same tag within return_list.

`<vertex_id_list>::=[vertex_id [, vertex_id]]` is a list of vertex IDs separated by comma (,).

`[YIELD [DISTINCT] <return_list>]` is the property list returned. Please refer YIELD Syntax for usage.

Examples

```
-- return all the properties of vertex 100.
nebula> FETCH PROP ON * 100;

-- return all the properties on tag player and team of vertex 100, 102
nebula> FETCH PROP ON * 100, 102;

-- return all properties of vertex 100, 201
nebula> FETCH PROP ON player, team 100, 201;

-- return all the properties in tag player of vertex id 100 if no yield field is given.
nebula> FETCH PROP ON player 100;

-- return property name and age of vertex id 100.
nebula> FETCH PROP ON player 100 YIELD player.name, player.age;

-- hash string to int64 as vertex id, fetch name and player.
nebula> FETCH PROP ON player hash("nebula")  YIELD player.name, player.age;

-- you can use fetch with pipe.
nebula> YIELD 100 AS id | FETCH PROP ON player $-.id;

-- find all neighbors of vertex 100 through edge follow. Then get the neighbors' name and age.
nebula> GO FROM 100 OVER follow YIELD follow._dst AS id | FETCH PROP ON player $-.id YIELD player.name, player.age;

-- the same as above statement.
nebula> $var = GO FROM 100 OVER follow YIELD follow._dst AS id; FETCH PROP ON player $var.id YIELD player.name, player.age;

-- get three vertices 100, 101, 102 and return by unique(distinct) name and age.
nebula> FETCH PROP ON player 100,101,102 YIELD DISTINCT player.name, player.age;
```

FETCH EDGE PROPERTY

The `FETCH` usage of an edge is almost the same with vertex. You can get properties from multiple edges with the same type.

```
FETCH PROP ON <edge_type> <vid> -> <vid>[@<rank>] [, <vid> -> <vid> ...] [YIELD [DISTINCT] <return_list>]
```

`<edge_type>` specifies the edge's type. It must be the same as those in `<return_list>`.

`<vid> -> <vid>` denotes a starting vertex to (->) an ending vertex. Multiple edges are separated by comma(,).

`<rank>` specifies the edge rank of the same edge type; it's optional. If not specified, the edge ranked 0 is returned by default.

`[YIELD [DISTINCT] <return_list>]` is the property list returned.

Example

```
-- from vertex 100 to 200 with edge type serve, get all the properties since no YIELD is given.
nebula> FETCH PROP ON serve 100 -> 200;

-- only return property start_year.
nebula> FETCH PROP ON serve 100 -> 200 YIELD serve.start_year;

-- for all the out going edges of vertex 100, get edge property degree.
```

```
nebula> GO FROM 100 OVER follow YIELD follow.degree;

-- the same as above statement.
nebula> GO FROM 100 OVER follow YIELD follow._src AS s, follow._dst AS d \
 | FETCH PROP ON follow $-.s -> $-.d YIELD follow.degree;

-- the same as above.
nebula> $var = GO FROM 100 OVER follow YIELD follow._src AS s, follow._dst AS d;\
 FETCH PROP ON follow $var.s -> $var.d YIELD follow.degree;
```

Last update: April 15, 2021

GO Syntax

`GO` statement is the MOST commonly used clause in **Nebula Graph**.

It indicates to traverse in a graph with specific filters (the `WHERE` clause), to fetch properties of vertices and edges, and return results (the `YIELD` clause) with given order (the `ORDER BY ASC | DESC` clause) and numbers (the `LIMIT` clause).

> The syntax of `GO` statement is very similar to `SELECT` in SQL. Notice that the major difference is that `GO` must start traversing from a (set of) vertex (vertices).

```
  GO [[<M> TO] <N> STEPS ] FROM <node_list>
  OVER <edge_type_list> [REVERSELY] [BIDIRECT]
  [ WHERE <expression> [ AND | OR expression ...]) ]
  YIELD [DISTINCT] <return_list>

<node_list>
   | <vid> [, <vid> ...]
   | $-.id

<edge_type_list>
   edge_type [, edge_type ...]
   * # `*` selects all the available edge types

<return_list>
    <col_name> [AS <col_alias>] [, <col_name> [AS <col_alias>] ...]
```

- `<N> STEPS` specifies the N query hops. If not specified, the default traversal is one hop. When `N` is zero, **Nebula Graph** will not traverse any edges, so the returned result is empty.
- `M TO N STEPS` traverses from M to N hops. When `M` is zero, the return result is the same to `M` is one. That is, the return result of `GO 0 TO 2` and `GO 1 TO 2` are the same.
- `<node_list>` is either a list of node's vid separated by comma(,), or a special place holder `$-.id` (refer `PIPE` syntax).
- `<edge_type_list>` is a list of edge types which graph traversal can go through.
- `WHERE <expression>` extracts only those results that fulfill the specified conditions. WHERE syntax can be conditions for src-vertex, the edges, and dst-vertex. The logical AND, OR, NOT are also supported. See WHERE Syntax for more information.
- `YIELD [DISTINCT] <return_list>` statement returns the result in column format and rename as an alias name. See `YIELD` syntax for more information. The `DISTINCT` syntax works the same as SQL.

EXAMPLES

```
nebula> GO FROM 107 OVER serve;  \
   /* start from vertex 107 along with edge type serve, and get vertex 200, 201 */
==============
| serve._dst |
==============
| 200        |
--------------
| 201        |
--------------
```

```
nebula> GO 2 STEPS FROM 103 OVER follow; \
  /* return the 2 hop friends of the vertex 103 */
===============
| follow._dst |
===============
| 101         |
---------------
```

```
nebula> GO FROM 109 OVER serve  \
   WHERE serve.start_year > 1990       /* check edge (serve) property ( start_year) */ \
   YIELD $$.team.name AS team_name, serve.start_year as start_year;   /* target vertex (team) property serve.start_year */
=========================
| team_name | start_year |
=========================
| Nuggets   | 2011       |
-------------------------
| Rockets   | 2017       |
-------------------------
```

```
nebula> GO FROM 100,102 OVER serve           \
        WHERE serve.start_year > 1995            /* check edge property */ \
        YIELD DISTINCT $$.team.name AS team_name, /* DISTINCT as SQL */ \
        serve.start_year as start_year,          /* edge property */ \
        $^.player.name AS player_name            /* source vertex (player) property */
=============================================
| team_name | start_year | player_name       |
```

```
=============================================
| Warriors  | 2001       | LaMarcus Aldridge |
---------------------------------------------
| Warriors  | 1997       | Tim Duncan        |
---------------------------------------------
```

Traverse Along Multiple Edges Types

Currently, **Nebula Graph** also supports traversing via multiple edge types with `GO`. The syntax is:

```
GO FROM <node_list> OVER <edge_type_list | *> YIELD [DISTINCT] <return_list>
```

For example:

```
nebula> GO OVER FROM <node_list> edge1, edge2....  // traverse along edge1 and edge2 or
nebula> GO OVER FROM <node_list> *   // * indicates traversing along all edge types
```

NOTE: Please note that when traversing along multiple edges, there are some special restrictions on the use of filters(namely the `WHERE` statement), for example filters like `WHERE edge1.prop1 > edge2.prop2` is not supported.

As for return results, if multiple edge properties are to be returned, **Nebula Graph** will place them in different rows. For example:

```
nebula> GO FROM 100 OVER follow, serve YIELD follow.degree, serve.start_year;

The following result is returned:

====================================
| follow.degree | serve.start_year |
====================================
| 0             | 1997             |
------------------------------------
| 95            | 0                |
------------------------------------
| 89            | 0                |
------------------------------------
| 90            | 0                |
------------------------------------

If there is no property, the default value will be placed. The default value for numeric type is 0, and for string type is an empty string, for bool is
false, for timestamp is 0 (namely “1970-01-01 00:00:00”) and for double is 0.0.

Of course, you can query without specifying `YIELD`, which returns the vids of the destination vertices of each edge. Again, default values (here is 0) will
be placed if there is no property. For example, query `GO FROM 100 OVER follow, serve;` returns the follow lines:

============================
| follow._dst | serve._dst |
============================
| 0           | 200        |
----------------------------
| 101         | 0          |
----------------------------
| 102         | 0          |
----------------------------
| 106         | 0          |
----------------------------
```

For query statement `GO FROM 100 OVER *`, the result is similar to the above example: the non-existing property or vid is populated with default values. Please note that we can't tell which row belongs to which edge in the results. The future version will show the edge type in the result.

TRAVERSE REVERSELY

Currently, **Nebula Graph** supports traversing reversely using keyword `REVERSELY`. The syntax is:

```
  GO FROM <node_list>
  OVER <edge_type_list> REVERSELY
  WHERE (expression [ AND | OR expression ...])
  YIELD [DISTINCT] <return_list>
```

For example:

```
nebula> GO FROM 100 OVER follow REVERSELY YIELD follow._src; -- returns 100
```

```
nebula> GO FROM 100 OVER follow REVERSELY YIELD follow._dst AS id | \
        GO FROM $-.id OVER serve WHERE $^.player.age > 20 YIELD $^.player.name AS FriendOf, $$.team.name AS Team;

============================
| FriendOf      | Team     |
```

```
============================
| Tony Parker   | Warriors |
----------------------------
| Kyle Anderson | Warriors |
----------------------------
```

The above query first traverses players that follow player 100 and finds the teams they serve, then filter players who are older than 20, and finally it returns their names and teams. Of course, you can query without specifying YIELD, which will return the vids of the destination vertices of each edge by default.

TRAVERSE BIDIRECT

Currently, **Nebula Graph** supports traversing along in and out edges using keyword `BIDIRECT`, the syntax is:

```
  GO FROM <node_list>
  OVER <edge_type_list> BIDIRECT
  WHERE (expression [ AND | OR expression ...])
  YIELD [DISTINCT] <return_list>
```

For example:

```
nebula> GO FROM 102 OVER follow BIDIRECT;
===============
| follow._dst |
===============
| 101         |
---------------
| 103         |
---------------
| 135         |
---------------
```

The above query returns players followed by 102 and follow 102 at the same time.

TRAVERSE FROM M TO N HOPS

Nebula Graph supports traversing from M to N hops. When M is equal to N, `GO M TO N STEPS` is equivalent to `GO N STEPS`. The syntax is:

```
  GO <M> TO <N> STEPS FROM <node_list>
  OVER <edge_type_list> [REVERSELY] [BIDIRECT]
  [YIELD [DISTINCT] <return_list>]
```

For example:

```
nebula> GO 1 TO 2 STEPS FROM 100 OVER serve;
==============
| serve._dst |
==============
| 200        |
--------------
```

Traverse from vertex 100 along edge type serve, return 1 to 2 hops.

```
nebula> GO 2 TO 4 STEPS FROM 100 OVER follow REVERSELY YIELD DISTINCT follow._dst;
===============
| follow._dst |
===============
| 133         |
---------------
| 105         |
---------------
| 140         |
---------------
```

Traverse from vertex 100 along edge type follow reversely, return 2 to 4 hops.

```
nebula> GO 4 TO 5 STEPS FROM 101 OVER follow BIDIRECT YIELD DISTINCT follow._dst;
===============
| follow._dst |
===============
| 100         |
---------------
| 102         |
---------------
| 104         |
---------------
| 105         |
---------------
```

```
| 107         |
---------------
| 113         |
---------------
| 121         |
---------------
```

Traverse from vertex 101 along edge type follow bi-directly, return 4 to 5 hops.

PASSING INT TYPE TO GO QUERIES

```
... | GO FROM $-.id OVER <edge_type_list>
```

For example:

```
nebula> YIELD 100 AS id | GO FROM $-.id OVER serve;
==============
| serve._dst |
==============
| 200        |
--------------
```

Last update: April 8, 2021

INSERT EDGE Syntax

```
INSERT EDGE <edge_name> ( <prop_name_list> ) VALUES | VALUE
<src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> )
[, <src_vid> -> <dst_vid> : ( <prop_value_list> ), ...]

<prop_name_list> ::=
  [ <prop_name> [, <prop_name> ] ...]

<prop_value_list> ::=
  [ <prop_value> [, <prop_value> ] ...]
```

`INSERT EDGE` statement inserts a (directed) edge from a starting vertex (given by src_vid) to an ending vertex (given by dst_vid).

- `<edge_name>` denotes the edge type, which must be created before `INSERT EDGE`.
- `<prop_name_list>` is the property name list as the given `<edge_name>`.
- `<prop_value_list>` must provide the value list according to `<prop_name_list>`. If no value matches the type, an error will be returned.
- `rank` is optional, it specifies the edge rank of the same edge type, if not specified, the default value is 0.

EXAMPLES

```
nebula> CREATE EDGE e1()                    -- create edge t1 with empty property or default values
nebula> INSERT EDGE e1 () VALUES 10->11:()    -- insert an edge from vertex 10 to vertex 11 with empty property
nebula> INSERT EDGE e1 () VALUES 10->11@1:()  -- insert an edge from vertex 10 to vertex 11 with empty property, the edge rank is 1
```

```
nebula> CREATE EDGE e2 (name string, age int)                    -- create edge e2 with two properties
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 1)          -- insert edge from 11 to 13 with two properties
nebula> INSERT EDGE e2 (name, age) VALUES \
12->13:("n1", 1), 13->14:("n2", 2)                           -- insert two edges
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", "a13")      -- ERROR. "a13" is not int
```

An edge can be inserted/wrote multiple times. Only the last written values can be read.

```
-- insert edge with the new values.
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 12)
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 13)
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 14) -- the last version can be read
```

Last update: April 8, 2021

INSERT VERTEX Syntax

```
INSERT VERTEX <tag_name> (prop_name_list) [, <tag_name> (prop_name_list), ...]
     {VALUES | VALUE} VID: (prop_value_list[, prop_value_list])

prop_name_list:
  [prop_name [, prop_name] ...]

prop_value_list:
  [prop_value [, prop_value] ...]
```

The `INSERT VERTEX` statement inserts a vertex or vertices into **Nebula Graph**.

- `tag_name` denotes the `tag` (vertex type), which must be created before `INSERT VERTEX`.
- `prop_name_list` is the property name list in the given `tag_name`.
- `VID` is the vertex ID. The `VID` must be unique in the graph space. The current sorting basis is "binary coding order", i.e. 0, 1, 2, ... 9223372036854775807, -9223372036854775808, -9223372036854775807, ..., -1. `VID` supports specifying ID manually, or call hash() function to generate.
- `prop_value_list` must provide the value list according to the `prop_name_list`. If no value matches the type, an error will be returned.

EXAMPLES

```
nebula> CREATE TAG t1()                   -- create tag t1 with empty property
nebula> INSERT VERTEX t1 () VALUES 10:()    -- insert vertex 10 with no property
```

```
nebula> CREATE TAG t2 (name string, age int)               -- create tag t2 with two properties
nebula> INSERT VERTEX t2 (name, age) VALUES 11:("n1", 12)     -- insert vertex 11 with two properties
nebula> INSERT VERTEX t2 (name, age) VALUES 12:("n1", "a13")  -- ERROR. "a13" is not int
nebula> INSERT VERTEX t2 (name, age) VALUES 13:("n3", 12), 14:("n4", 8)    -- insert two vertices
```

```
nebula> CREATE TAG t1(i1 int)
nebula> CREATE TAG t2(s2 string)
nebula> INSERT VERTEX  t1 (i1), t2(s2) VALUES 21: (321, "hello")   -- insert vertex 21 with two tags.
```

A vertex can be inserted/wrote multiple times. Only the last written values can be read.

```
-- insert vertex 11 with the new values.
nebula> INSERT VERTEX t2 (name, age) VALUES 11:("n2", 13)
nebula> INSERT VERTEX t2 (name, age) VALUES 11:("n3", 14)
nebula> INSERT VERTEX t2 (name, age) VALUES 11:("n4", 15)  -- the last version can be read
```

Last update: April 8, 2021

LOOKUP Syntax

The `LOOKUP` statement is used to search for the filter condition in it. `LOOKUP` is often coupled with a `WHERE` clause which adds filters or predicates.

> **NOTE**: Before using the `LOOKUP` statement, please make sure that indexes are created. Read more about indexes in Index Documentation.

```
LOOKUP ON {<vertex_tag> | <edge_type>} WHERE <expression> [ AND expression ...]) ] [YIELD <return_list>]

<return_list>
    <col_name> [AS <col_alias>] [, <col_name> [AS <col_alias>] ...]
```

- `LOOKUP` clause finds the vertices or edges.
- `WHERE` extracts only those results that fulfill the specified conditions. Only logical AND is supported. See WHERE Syntax for more information.
- `YIELD` clause returns particular results. If not specified, vertex ID is returned when `LOOKUP` tags, source vertex ID, destination vertex ID and ranking of the edges are returned when `LOOKUP` edges.

RESTRICTIONS FOR INDEX USAGE

The `WHERE` clause does not support the following operations in `LOOKUP`:

- `$-` and `$^`
- In relational expressions, expressions with field-names on both sides of the operator are not currently supported, such as (tagName.column1> tagName.column2)
- Nested AliasProp expressions in operation expressions and function expressions are not supported at this time.
- Range scan is not supported in the string type index.
- The `OR` and the `OXR` operations are not supported.

RETRIEVE VERTICES

The following example returns vertices whose name is `Tony Parker` and tagged with *player*.

```
nebula> CREATE TAG INDEX index_player ON player(name, age);

nebula> LOOKUP ON player WHERE player.name == "Tony Parker";
============
| VertexID |
============
| 101      |
------------

nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
YIELD player.name, player.age;
======================================
| VertexID | player.name | player.age |
======================================
| 101      | Tony Parker | 36         |
--------------------------------------

nebula> LOOKUP ON player WHERE player.name== "Kobe Bryant" YIELD player.name AS name | \
GO FROM $-.VertexID OVER serve YIELD $-.name, serve.start_year, serve.end_year, $$.team.name;
=================================================================
| $-.name     | serve.start_year | serve.end_year | $$.team.name |
=================================================================
| Kobe Bryant | 1996             | 2016           | Lakers       |
-----------------------------------------------------------------
```

RETRIEVE EDGES

The following example returns edges whose `degree` is 90 and the edge type is *follow*.

```
nebula> CREATE EDGE INDEX index_follow ON follow(degree);

nebula> LOOKUP ON follow WHERE follow.degree == 90;
============================
| SrcVID | DstVID | Ranking |
============================
| 100    | 106    | 0       |
----------------------------

nebula> LOOKUP ON follow WHERE follow.degree == 90 YIELD follow.degree;
```

```
==========================================
| SrcVID | DstVID | Ranking | follow.degree |
==========================================
| 100    | 106    | 0       | 90            |
------------------------------------------

nebula> LOOKUP ON follow WHERE follow.degree == 90 YIELD DISTINCT follow._src;
=======================================
| SrcVID | DstVID | Ranking | follow._src |
=======================================
| 121    | 116    | 0       | 121         |
---------------------------------------
| 140    | 114    | 0       | 140         |
---------------------------------------
| 142    | 117    | 0       | 142         |
---------------------------------------
| 133    | 114    | 0       | 133         |
---------------------------------------
| 143    | 150    | 0       | 143         |
---------------------------------------
| 114    | 103    | 0       | 114         |
---------------------------------------
| 136    | 117    | 0       | 136         |
---------------------------------------
| 127    | 114    | 0       | 127         |
---------------------------------------
| 147    | 136    | 0       | 147         |
---------------------------------------
| 118    | 120    | 0       | 118         |
---------------------------------------
| 128    | 116    | 0       | 128         |
---------------------------------------
| 138    | 115    | 0       | 138         |
---------------------------------------
| 129    | 116    | 0       | 129         |
---------------------------------------

nebula> LOOKUP ON follow WHERE follow.degree == 90 YIELD DISTINCT follow._dst;
=======================================
| SrcVID | DstVID | Ranking | follow._dst |
=======================================
| 121    | 116    | 0       | 116         |
---------------------------------------
| 140    | 114    | 0       | 114         |
---------------------------------------
| 142    | 117    | 0       | 117         |
---------------------------------------
| 133    | 114    | 0       | 114         |
---------------------------------------
| 114    | 103    | 0       | 103         |
---------------------------------------
| 136    | 117    | 0       | 117         |
---------------------------------------
| 118    | 120    | 0       | 120         |
---------------------------------------
| 128    | 116    | 0       | 116         |
---------------------------------------

nebula> LOOKUP ON follow WHERE follow.degree == 60 YIELD follow.degree AS Degree | \
GO FROM $-.DstVID OVER serve YIELD $-.DstVID, serve.start_year, serve.end_year, $$.team.name;
================================================================
| $-.DstVID | serve.start_year | serve.end_year | $$.team.name |
================================================================
| 105       | 2010             | 2018           | Spurs        |
----------------------------------------------------------------
| 105       | 2009             | 2010           | Cavaliers    |
----------------------------------------------------------------
| 105       | 2018             | 2019           | Raptors      |
----------------------------------------------------------------
```

FAQ

Error code 411

```
[ERROR (-8)]: Unknown error(411):
```

Error code `411` shows there is no valid index for the current `WHERE` filter. Nebula Graph uses the left matching mode to select indexes. That is, columns in the `WHERE` filter must be in the first N columns of the index. For example:

```
nebula> CREATE TAG INDEX example_index ON TAG t(p1, p2, p3);  -- Create an index for the first 3 properties of tag t
nebula> LOOKUP ON t WHERE p2 == 1 and p3 == 1; -- Not supported
nebula> LOOKUP ON t WHERE p1 == 1;  -- Supported
nebula> LOOKUP ON t WHERE p1 == 1 and p2 == 1;  -- Supported
nebula> LOOKUP ON t WHERE p1 == 1 and p2 == 1 and p3 == 1;  -- Supported
```

No valid index found

```
No valid index found
```

If your query filter contains a string type field, Nebula Graph selects the index that matches all the fields. For example:

```
nebula> CREATE TAG t1 (c1 string, c2 int);
nebula> CREATE TAG INDEX i1 ON t1 (c1, c2);
nebula> LOOKUP ON t1 WHERE t1.c1 == "a"; -- Index i1 is invalid
nebula> LOOKUP ON t1 WHERE t1.c1 == "a" and t1.c2 == 1;  -- Index i1 is valid
```

Last update: April 16, 2021

RETURN Syntax

The `RETURN` statement is used to return the result when the condition is true. If the condition is false, no result is returned.

```
    RETURN <var_ref> IF <var_ref> IS NOT NULL
```

- is a variable name, e.g. `$var`.

EXAMPLES

```
nebula> $A = GO FROM 100 OVER follow YIELD follow._dst AS dst; \
        $rA = YIELD $A.* WHERE $A.dst == 101; \
        RETURN $rA IF $rA is NOT NULL; \ /* Returns the result because $rA is not empty */
        GO FROM $A.dst OVER follow; /* As the RETURN statement returns the result, the GO FROM statement is not executed*/
==========
| $A.dst |
==========
| 101    |
----------
nebula> $A = GO FROM 100 OVER follow YIELD follow._dst AS dst; \
        $rA = YIELD $A.* WHERE $A.dst == 300; \
        RETURN $rA IF $rA is NOT NULL; \ /* Does not return the result because $rA is empty */
        GO FROM $A.dst OVER follow; /* As the RETURN statement does not return the result, the GO FROM statement is executed */
===============
| follow._dst |
===============
| 100         |
---------------
| 101         |
---------------
| 100         |
---------------
| 102         |
---------------
| 100         |
---------------
| 107         |
---------------
```

Last update: April 8, 2021

UPDATE EDGE Syntax

Nebula Graph supports `UPDATE EDGE` properties of an edge, as well as CAS operation and returning related properties. The `UPDATE EDGE` statement only updates one edge-type of an edge at a time.

```
UPDATE EDGE <edge> SET <update_columns> [WHEN <condition>] [YIELD <columns>]
```

NOTE: `WHEN` and `YIELD` are optional.

- `edge` is the edge to be updated, the syntax is `<src> -> <dst> [@rank] OF <edge_type>`.
- `update_columns` is the properties of the edge to be updated.
- `condition` is some constraints, only when met, `UPDATE` will run successfully and expression operations are supported.
- `columns` is the columns to be returned, `YIELD` returns the latest updated values.

Consider the following example:

```
nebula> UPDATE EDGE 100 -> 200@0 OF serve SET start_year = serve.start_year + 1 \
YIELD $^.player.name AS name, serve.start_year AS start;
```

Last update: April 8, 2021

UPDATE VERTEX Syntax

Nebula Graph supports `UPDATE VERTEX` properties of a vertex, as well as CAS operation and returning related properties. The `UPDATE VERTEX` statement only updates one tag of a vertex at a time.

```
UPDATE VERTEX <vid> SET <update_columns> [WHEN <condition>] [YIELD <columns>]
```

NOTE: `WHEN` and `YIELD` are optional.

- `vid` is the id of the vertex to be updated.
- `update_columns` is the properties of the vertex to be updated, for example, `tag1.col1 = $^.tag2.col2 + 1` means to update `tag1.col1` to `tag2.col2+1`.

  NOTE: `$^` indicates vertex to be updated.
- `condition` is some constraints, only when met, `UPDATE` will run successfully and expression operations are supported.
- `columns` is the columns to be returned, `YIELD` returns the latest updated values.

Consider the following example:

```
nebula> UPDATE VERTEX 101 SET player.age = $^.player.age + 1 \
WHEN $^.player.name == "Tony Parker" \
YIELD $^.player.name AS name, $^.player.age AS age;
```

There are one tag in vertex 101, namely player.

```
nebula> UPDATE VERTEX 200 SET player.name = 'Cory Joseph' WHEN $^.team.name == 'Rocket';
[ERROR (-8)]: Maybe invalid tag or property in SET/YIELD clause!
```

`UPDATE VERTEX` does not support multiple tags, so an error occurs here.

Last update: April 8, 2021

UPSERT Syntax

`UPSERT` is used to insert a new vertex or edge or update an existing one. If the vertex or edge doesn't exist it will be created. `UPSERT` is a combination of `INSERT` and `UPDATE`.

The performance of `UPSERT` is much lower than that of `INSERT`, because `UPSERT` is a read-modify-write serialization operation at the partition level. So it is not suitable for large concurrent write scenarios.

- If the vertex or edge does not exist, a new one will be created regardless of whether the condition in WHEN clause is met. The property columns not specified by the `SET` statement use the default values of the columns, if there are no default values, an error will be returned;
- If the vertex or edge exists and the WHEN condition is met, the vertex or edge will be updated;
- If the vertex or edge exists and the WHEN condition is not met, nothing will be done.

```
UPSERT {VERTEX <vid> | EDGE <edge>} SET <update_columns> [WHEN <condition>] [YIELD <columns>]
```

- `vid` is the ID of the vertex to be updated.
- `edge` is the edge to be updated, the syntax is `<src> -> <dst> [@rank] OF <edge_type>`.
- `update_columns` is the properties of the vertex or edge to be updated, for example, `tag1.col1 = $^.tag2.col2 + 1` means to update `tag1.col1` to `tag2.col2+1`.

  NOTE: `$^` indicates vertex to be updated.
- `condition` is some constraints, only when met, `UPSERT` will run successfully and expression operations are supported.
- `columns` is the columns to be returned, `YIELD` returns the latest updated values.

Consider the following example:

```
nebula> INSERT VERTEX player(name, age) VALUES 111:("Ben Simmons", 22); -- Insert a new vertex.
nebula> UPSERT VERTEX 111 SET player.name = "Dwight Howard", player.age = $^.player.age + 11 WHEN $^.player.name == "Ben Simmons" && $^.player.age > 20 YIELD
$^.player.name AS Name, $^.player.age AS Age; -- Do upsert on the vertex.
======================
| Name          | Age |
======================
| Dwight Howard | 33  |
----------------------
```

```
nebula> FETCH PROP ON * 111; -- An empty set is returned, indicating vertex 111 does not exist.
Empty set (Time spent: 3.069/4.382 ms)
nebula> UPSERT VERTEX 111 SET player.age = $^.player.age + 1;
```

When vertex 111 does not exist and the player's age has a default value, the player.age of vertex 111 is the default value + 1. If player.age does not have default value, an error will be reported.

```
nebula> CREATE TAG person(followers int, age int DEFAULT 0); -- Create example tag person

nebula> UPSERT VERTEX 300 SET person.followers = $^.person.age + 1,  person.age = 8; -- followers is 1, age is 8

nebula> UPSERT VERTEX 300 SET person.age = 8, person.followers = $^.person.age + 1; -- followers is 9, age is 8
```

Last update: April 8, 2021

WHERE Syntax

The `WHERE` clause allows you to specify a search condition for the data returned by a query. The following shows the syntax of the WHERE clause:

```
WHERE <expression> [ AND | OR <expression> ...])
```

Currently, the `WHERE` statement applies to the `GO` and `LOOKUP` statement. Note some `WHERE` filter conditions are not supported in the `LOOKUP` statement. Refer to the LOOKUP Doc for details.

Usually, `WHERE` is a set of logical combination that filters vertex or edge properties.

> As syntactic sugar, logic AND is represented by `AND` or `&&` and logic OR is represented by `OR` or `||`.

EXAMPLES

```
-- the degree property of edge follow is greater than 90.
nebula> GO FROM 100 OVER follow WHERE follow.degree > 90;
-- the following result is returned:
===============
| follow._dst |
===============
| 101         |
---------------

-- find the destination vertex whose age is equal to the source vertex, player 104.
nebula> GO FROM 104 OVER follow WHERE $^.player.age == $$.player.age;
-- the following result is returned:
===============
| follow._dst |
===============
| 103         |
---------------

-- logical combination is allowed.
nebula> GO FROM 100 OVER follow WHERE follow.degree > 90 OR $$.player.age != 33 AND $$.player.name != "Tony Parker";
-- the following result is returned:
===============
| follow._dst |
===============
| 101         |
---------------
| 106         |
---------------

-- the condition in the WHERE clause is always TRUE.
nebula> GO FROM 101 OVER follow WHERE 1 == 1 OR TRUE;
-- the following result is returned:
===============
| follow._dst |
===============
| 100         |
---------------
| 102         |
---------------
```

FILTERING EDGE RANK WITH WHERE

You can filter the edge rank with the `WHERE` clause. For example:

```
nebula> CREATE SPACE test;
nebula> USE test;
nebula> CREATE EDGE e1(p1 int);
nebula> CREATE TAG person(p1 int);
nebula> INSERT VERTEX person(p1) VALUES 1:(1);
nebula> INSERT VERTEX person(p1) VALUES 2:(2);
nebula> INSERT EDGE e1(p1) VALUES 1->2@0:(10);
nebula> INSERT EDGE e1(p1) VALUES 1->2@1:(11);
nebula> INSERT EDGE e1(p1) VALUES 1->2@2:(12);
nebula> INSERT EDGE e1(p1) VALUES 1->2@3:(13);
nebula> INSERT EDGE e1(p1) VALUES 1->2@4:(14);
nebula> INSERT EDGE e1(p1) VALUES 1->2@5:(15);
nebula> INSERT EDGE e1(p1) VALUES 1->2@6:(16);
nebula> GO FROM 1 OVER e1 WHERE e1._rank>2 YIELD e1._src, e1._dst, e1._rank AS Rank, e1.p1 | ORDER BY Rank DESC;
==================================
| e1._src | e1._dst | Rank | e1.p1 |
==================================
| 1       | 2       | 6    | 16    |
----------------------------------
| 1       | 2       | 5    | 15    |
----------------------------------
| 1       | 2       | 4    | 14    |
----------------------------------
```

```
| 1      | 2      | 3    | 13    |
-----------------------------------
```

Last update: April 8, 2021

YIELD Syntax

Keyword `YIELD` can be used as a clause in a `FETCH` or `GO` statement, or as a separate statement in `PIPE` (`|`), or as a stand-alone statement for calculation.

AS CLAUSE (WITH GO-SYNTAX)

```
YIELD
    [DISTINCT]
    <col_name> [AS <col_alias>]
    [, <col_name> [AS <col_alias>] ...]
```

`YIELD` is commonly used to return results generated with `GO` (Refer GO).

```
nebula> GO FROM 100 OVER follow YIELD $$.player.name AS Friend, $$.player.age AS Age;
==========================
| Friend            | Age |
==========================
| Tony Parker       | 36  |
--------------------------
| LaMarcus Aldridge | 33  |
--------------------------
| Kyle Anderson     | 25  |
--------------------------
```

For example: `$$.player.name` is used to get the property of the destination vertex ($$).

AS STATEMENT

Reference Inputs or Variables

- You can use the `YIELD` statement in `PIPE`.
- You can use the `YIELD` statement to reference variables.
- For statements that do not support `YIELD` statement, you can use it as a tool to control the output.

```
YIELD
    [DISTINCT]
    <col_name> [AS <col_alias>]
    [, <col_name> [AS <col_alias>] ...]
    [WHERE <conditions>]
```

```
nebula> GO FROM 100 OVER follow YIELD follow._dst AS id | YIELD $-.* WHERE $-.id == 106;

=========
| $-.id |
=========
| 106   |
---------

nebula> $var1 = GO FROM 101 OVER follow; $var2 = GO FROM 105 OVER follow; YIELD $var1.* UNION YIELD DISTINCT $var2.*;

====================
| $var1.follow._dst |
====================
| 100               |
--------------------
| 102               |
--------------------
| 104               |
--------------------
| 116               |
--------------------
| 125               |
--------------------

nebula> GO 2 STEPS FROM 100 OVER follow YIELD follow._dst AS dst | YIELD DISTINCT $-.dst AS dst

=======
| dst |
=======
| 100 |
-------
| 102 |
-------
| 125 |
-------
```

As Stand-alone Statement

`YIELD` statement can be used independently to retrieve computation results without reference to any graph. You can use `AS` to rename it an alias.

```
nebula> YIELD 1 + 1;
=========
| (1+1) |
=========
| 2     |
---------

nebula> YIELD "Hel" + "\tlo" AS HELLO_1, ", World!" AS WORLD_2;
======================
| HELLO_1 | WORLD_2  |
======================
| Hel  lo  | , World! |
----------------------

nebula> YIELD hash("Tim") % 100;
====================
| (hash("Tim")%100) |
====================
| 42                |
--------------------
```

Last update: April 16, 2021

2.5.3 Utility Statements

SHOW Statements

SHOW CHARSET SYNTAX

```
SHOW CHARSET
```

`SHOW CHARSET` displays the available character sets. Currently available types are: utf8 and utf8mb4. The default charset type is utf8. **Nebula Graph** extends the uft8 to support four byte characters. Therefore utf8 and utf8mb4 equivalent.

```
nebula> SHOW CHARSET;
=============================================================
| Charset | Description   | Default collation | Maxlen |
=============================================================
| utf8    | UTF-8 Unicode | utf8_bin          | 4      |
-------------------------------------------------------------
```

`SHOW CHARSET` output has these columns:

- Charset The character set name.
- Description A description of the character set.
- Default collation The default collation for the character set.
- Maxlen The maximum number of bytes required to store one character.

Last update: April 8, 2021

SHOW COLLATION SYNTAX

```
SHOW COLLATION
```

`SHOW COLLATION` displays the collations supported by **Nebula Graph**. Currently available types are: utf8_bin, utf8_general_ci, utf8mb4_bin and utf8mb4_general_ci. When the character set is utf8, the default collate is utf8_bin; when the character set is utf8mb4, the default collate is utf8mb4_bin. Both utf8_general_ci and utf8mb4_general_ci are case-insensitive comparisons and behave the same as MySQL.

```
nebula> SHOW COLLATION;
=======================
| Collation | Charset |
=======================
| utf8_bin  | utf8    |
-----------------------
```

`SHOW COLLATION` output has these columns:

- Collation The collation name.
- Charset The name of the character set with which the collation is associated.

Last update: April 8, 2021

SHOW CONFIGS SYNTAX

```
SHOW CONFIGS [graph|meta|storage]
```

`SHOW CONFIGS` lists the configuration information. `SHOW CONFIGS` output has these columns: module, name, type, mode and value.

For example:

```
nebula> SHOW CONFIGS graph;
============================================================
| module | name                     | type  | mode    | value |
============================================================
| GRAPH  | v                        | INT64 | MUTABLE | 0     |
------------------------------------------------------------
| GRAPH  | minloglevel              | INT64 | MUTABLE | 2     |
------------------------------------------------------------
| GRAPH  | slow_op_threshhold_ms    | INT64 | MUTABLE | 50    |
------------------------------------------------------------
| GRAPH  | heartbeat_interval_secs | INT64 | MUTABLE | 3     |
------------------------------------------------------------
| GRAPH  | meta_client_retry_times | INT64 | MUTABLE | 3     |
------------------------------------------------------------
```

For more information about `SHOW CONFIGS [graph|meta|storage]`, please refer to configs syntax.

Last update: April 8, 2021

SHOW CREATE SPACE SYNTAX

```
SHOW CREATE SPACE <space_name>
```

`SHOW CREATE SPACE` statement returns the specified graph space and its creation syntax. If the graph space contains a default value, the default value is also returned.

```
nebula> SHOW CREATE SPACE basketballplayer;
=============================================================================================================
| Space | Create Space                                                                                      |
=============================================================================================================
| basketballplayer  | CREATE SPACE gods (partition_num = 1, replica_factor = 1, charset = utf8, collate = utf8_bin) |
-------------------------------------------------------------------------------------------------------------
```

Last update: April 16, 2021

SHOW CREATE TAGS/EDGES SYNTAX

```
SHOW CREATE {TAG <tag_name> | EDGE <edge_name>}
```

`SHOW CREATE TAG` and `SHOW CREATE EDGE` return the specified tag or edge type and their creation syntax in a given space. If the tag or edge type contains a default value, the default value is also returned.

```
nebula> SHOW CREATE TAG player;
==========================================================================================
| Tag    | Create Tag                                                                    |
==========================================================================================
| player | CREATE TAG player (
  name string,
  age int
) ttl_duration = 0, ttl_col = "" |
------------------------------------------------------------------------------------------
```

Last update: April 8, 2021

SHOW HOSTS SYNTAX

```
SHOW HOSTS
```

`SHOW HOSTS` statement lists storage hosts registered by the meta server. `SHOW HOSTS` output has these columns:: ip, port, status (online/offline), leader count, leader distribution, partition distribution.

```
nebula> SHOW HOSTS;
=============================================================================================
| Ip         | Port  | Status | Leader count | Leader distribution  | Partition distribution |
=============================================================================================
| 172.28.2.1 | 44500 | online | 9            | basketballplayer: 9  | basketballplayer: 10   |
---------------------------------------------------------------------------------------------
| 172.28.2.2 | 44500 | online | 0            |                      | basketballplayer: 10   |
---------------------------------------------------------------------------------------------
| 172.28.2.3 | 44500 | online | 1            | basketballplayer: 1  | basketballplayer: 10   |
---------------------------------------------------------------------------------------------
| Total      |       |        | 10           | basketballplayer: 10 | basketballplayer: 30   |
---------------------------------------------------------------------------------------------
```

Last update: April 16, 2021

SHOW INDEXES SYNTAX

```
SHOW {TAG | EDGE} INDEXES
```

`SHOW INDEXES` returns the defined tag/edg-type index information. `SHOW INDEXES` returns the following fields: index ID and index name.

For example:

```
nebula> SHOW TAG INDEXES;
============================
| Index ID | Index Name     |
============================
| 6        | player_index_1 |
----------------------------
| 7        | player_index_0 |
----------------------------
```

See Index on how to create indexes.

Last update: April 8, 2021

SHOW PARTS SYNTAX

```
SHOW PARTS <part_id>
```

`SHOW PARTS` lists the partition information of the given SPACE. `<part_id>` is optional, if not specified, all parts information is returned.

```
nebula> SHOW PARTS 1;
=============================================================
| Partition ID | Leader           | Peers            | Losts |
=============================================================
| 1            | 172.28.2.2:44500 | 172.28.2.2:44500 |       |
-------------------------------------------------------------
```

`SHOW PARTS` output has these columns:

- Partition ID
- Leader
- Peers
- Losts

Last update: April 8, 2021

SHOW ROLES SYNTAX

```
SHOW ROLES IN <space_name>
```

`SHOW ROLES` statement displays the roles that are assigned to a user account. `SHOW ROLES` output has these columns: account and role type.

If the user is `GOD` or `ADMIN`, **Nebula Graph** shows all roles limited to its authorized space. If the user is `DBA`, `USER` or `GUEST`, **Nebula Graph** shows only his own role.

For example:

```
nebula> SHOW ROLES in basketballplayer;
======================
| Account | Role Type |
======================
| userA   | ADMIN     |
----------------------
```

See Create User to create user. See Grant Role to grant roles to a user.

Last update: April 16, 2021

SHOW SNAPSHOTS SYNTAX

```
SHOW SNAPSHOTS
```

`SHOW SNAPSHOTS` statement lists all the snapshots.

For example:

```
nebula> SHOW SNAPSHOTS;
=============================================================
| Name                          | Status | Hosts           |
=============================================================
| SNAPSHOT_2019_12_04_10_54_36 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_42 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_44 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------------
```

See here to create snapshots.

Last update: April 8, 2021

SHOW SPACES SYNTAX

```
SHOW SPACES
```

`SHOW SPACES` lists the SPACES on the **Nebula Graph** cluster.

For example:

```
nebula> SHOW SPACES;
====================
| Name             |
====================
| basketballplayer |
--------------------
```

See here to creat spaces.

Last update: April 16, 2021

SHOW TAGS/EDGES SYNTAX

```
SHOW {TAGS | EDGES}
```

`SHOW TAGS` and `SHOW EDGES` return the defined tags and edge types in a given space, respectively.

Last update: April 8, 2021

SHOW USERS SYNTAX

```
SHOW USERS
```

`SHOW USERS` lists the users information. `SHOW USERS` output has these columns: account names.

Last update: April 8, 2021

DESCRIBE Syntax

```
DESCRIBE SPACE <space_name>
DESCRIBE TAG <tag_name>
DESCRIBE EDGE <edge_name>
DESCRIBE {TAG | EDGE} INDEX <index_name>
```

The DESCRIBE keyword is used to obtain information about space, tag and edge structure.

Also notice that DESCRIBE is different from SHOW. Refer SHOW.

EXAMPLE

Obtain information about space.

```
nebula> DESCRIBE SPACE basketballplayer;
========================================================================
| ID |     Name                 | Partition number | Replica Factor |
========================================================================
|  1 |     basketballplayer     |             100  |              1 |
------------------------------------------------------------------------
```

Obtain information about tag in a given space.

```
nebula> DESCRIBE TAG player;
==================
| Field | Type   |
==================
| name  | string |
------------------
| age   | int    |
------------------
```

Obtain information about edge in a given space.

```
nebula> DESCRIBE EDGE serve;
====================
| Field      | Type |
====================
| start_year | int  |
--------------------
| end_year   | int  |
--------------------
```

Obtain information about the index.

```
nebula> DESCRIBE TAG INDEX player_index_0;
==================
| Field | Type   |
==================
| name  | string |
------------------
```

Last update: April 16, 2021

USE Syntax

```
USE <graph_space_name>
```

The `USE` statement tells **Nebula Graph** to use the named (graph) space as the current working space for subsequent statements. This statement requires some privileges.

The named space remains the default until the end of the session or another `USE` statement is issued:

```
nebula> USE space1;
-- Traverse in graph space1.
nebula> GO FROM 1 OVER edge1;
nebula> USE space2;
-- Traverse in graph space2. These vertices and edges have no relevance with space1.
nebula> GO FROM 2 OVER edge2;
-- Now you are back to space1. Hereafter, you can not read any data from space2.
nebula> USE space1;
```

Different from SQL, making a space as the working space prevents you from accessing other spaces. The only way to traverse in a new graph space is to switch by the `USE` statement.

> SPACES are `FULLY ISOLATED` from each other. Unlike SQL, which allows you to select two tables from different databases in one statement, in **Nebula Graph**, you can only touch one space at a time.

Last update: April 8, 2021

2.5.4 Graph Algorithm

FIND PATH Syntax

`FIND PATH` statement can be used to get the shortest path and all paths.

```
FIND SHORTEST | ALL PATH FROM <vertex_id_list> TO <vertex_id_list> OVER <edge_type_list> [UPTO <N> STEPS]
```

`SHORTEST` is the keyword to find the shortest path.

`ALL` is the keyword to find all paths.

`<vertex_id_list>::=[vertex_id [, vertex_id]]` is the vertex id list,multiple ids should be separated with commas, and `$-` and `$var` are supported.

`<edge_type_list>` is the edge type list, multiple edge types should be separated with commas, and `*` can be referred as all edge types.

`<N>` is hop number, and the default value is 5.

- When source and destination vertices are id lists, it means to find the shortest path from any source vertices to the destination vertices.
- There may be cycles when searching all paths.
- `FIND PATH` does not support searching with property filtering.
- `FIND PATH` does not support searching with specified direction.
- `FIND PATH` is sing process, so it hurts the memory resource.

EXAMPLES

Path is displayed as `id <edge_name, rank> id` in console.

```
nebula> FIND SHORTEST PATH FROM 100 to 200 OVER *;
============================
| _path_ |
============================
| 100 <serve,0> 200
----------------------------
```

```
nebula>FIND ALL PATH FROM 100 to 200 OVER *;
=============================================================================================================
| _path_ |
=============================================================================================================
| 100 < serve,0> 200
-------------------------------------------------------------------------------------------------------------
| 100 <follow,0> 101 < serve,0> 200
-------------------------------------------------------------------------------------------------------------
| 100 <follow,0> 102 < serve,0> 200
-------------------------------------------------------------------------------------------------------------
| 100 <follow,0> 106 < serve,0> 200
-------------------------------------------------------------------------------------------------------------
```

Last update: April 8, 2021

3. Build Develop and Administration

3.1 Build

3.1.1 Build From Source Code

Overview

We have tested building on various environments, including CentOS 6 to 8, Ubuntu 16.04 to 19.04, Fedora 28 to 30, GCC 7.1.0 to 9.2.0 and recent Clang++ and the devtoolset of Red Hat and CentOS. But due to the complexity of building environments, we still cannot guarantee that we have covered all kinds of situations. If any problem encountered, please fire an issue or open a pull request to let us know.

Requirements

The following are the configuration requirements for compiling **Nebula Graph**. For the configuration requirements of the operating environment, see here.

- CPU: x86_64
- Memory: 4GB at least
- Disk space: 10GB at least
- Linux: 2.6.32 or higher, check with `uname -r`
- glibc: 2.12 or higher, check with `ldd --version`
- GCC: 7.1.0 or higher, check with `g++ --version`
- CMake: 3.5.0 or higher, check with `cmake --version`
- Access to the Internet

Quick Steps to Build

INSTALLING DEPENDENCIES

Please note that it requires root privileges to install packages.

For CentOS, RedHat and Fedora users:

```
$ yum update
$ yum install -y make \
                 m4 \
                 git \
                 wget \
                 unzip \
                 xz \
                 readline-devel \
                 ncurses-devel \
                 zlib-devel \
                 gcc \
                 gcc-c++ \
                 cmake \
                 gettext \
                 curl \
                 redhat-lsb-core

# For CentOS 8+, RedHat 8+, and Fedora, you need to install libstdc++-static, libasan
$ yum install -y libstdc++-static libasan
```

For Debian and Ubuntu users:

```
$ apt-get update
$ apt-get install -y make \
                     m4 \
                     git \
                     wget \
                     unzip \
```

```
                    xz-utils \
                    curl \
                    lsb-core \
                    build-essential \
                    libreadline-dev \
                    ncurses-dev \
                    cmake \
                    gettext
```

For Arch and Gentoo users, you can definitely handle all of these on your own, right?

To make sure your GCC and CMake are in the right version:

```
$ g++ --version
$ cmake --version
```

If not, please refer to the following sections: Install an Applicable CMake and Install an Applicable GCC.

CLONING THE REPO

```
$ git clone https://github.com/vesoft-inc/nebula.git
```

If you don't care about the commit history of the repo, and to make the cloning faster, you could perform a shallow clone:

```
$ git clone --depth=1 https://github.com/vesoft-inc/nebula.git
```

CONFIGURING AND BUILDING

```
$ cd nebula
$ mkdir build
$ cd build
$ cmake -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=Release ..
# Assuming cores is the number of cores and mem_gb is the memory size (in GB), the value of N is recommended to select the smaller one from cores and
mem_gb / 2
# We suggest choosing release build type to speed up compilation
$ make -jN
# The default installation directory is /usr/local/nebula
$ sudo make install
# If you want to start the services, copy the configuration files under the etc/ directory
# For production
$ cd /usr/local/nebula
$ sudo cp etc/nebula-storaged.conf.production etc/nebula-storaged.conf
$ sudo cp etc/nebula-metad.conf.production etc/nebula-metad.conf
$ sudo cp etc/nebula-graphd.conf.production etc/nebula-graphd.conf
# For trial
$ cd /usr/local/nebula
$ sudo cp etc/nebula-storaged.conf.default etc/nebula-storaged.conf
$ sudo cp etc/nebula-metad.conf.default etc/nebula-metad.conf
$ sudo cp etc/nebula-graphd.conf.default etc/nebula-graphd.conf
```

See the Start and Stop Nebula Graph Services Doc for details.

Since C++ templates are heavily used by **Nebula Graph** and its third party dependencies, especially Folly, fbthrift and boost, the building is very time-consuming.

For your reference, it is expected to take about 35 minutes in CPU time(less than 4 minutes with `-j16`), given an Intel E5-2697 v3 processor and unit tests are disabled.

PACKING YOUR SOURCE CODE (OPTIONAL)

- Package **Nebula Graph** to one package with the following command:

```
cd nebula/package
./package.sh -v <version>
```

- Package **Nebula Graph** to multiple packages with the following command:

```
cd nebula/package
./package.sh -v <version> -n OFF
```

Ways to Tweak the Building

Until now, you might already have built **Nebula Graph** successfully. If so or not, we also provide ways to tweak the building process.

CMAKE ARGUMENTS/VARIABLES

We provide several options to make one tweak the building, while some of these are builtins from CMake. These arguments are used at the configure(cmake) stage, like `cmake -DArgument=Value ..`.

ENABLE_WERROR

By default, **Nebula Graph** turns on the `-Werror` compiler option to regard any warnings as errors. If the building fails with such errors, you could still continue the building by set `ENABLE_WERROR` to `OFF`.

ENABLE_TESTING

This option allows to enable or disable the build of unit tests. We suggest to turn it off if you just need the service modules of **Nebula Graph**. Options are `ON` and `OFF`, and the default value is `ON`.

ENABLE_ASAN

This option enables or disables the ASan building, a.k.a AddressSanitizer, which is a memory error detector. It is meant to be used by **Nebula Graph** developers. This option is `OFF` by default.

CMAKE_BUILD_TYPE

There are a few building types supported:

- `Debug`, to build with debug info but without optimization, which is by default
- `Release`, to build with optimization but without debug info
- `RelWithDebInfo`, to build with optimization AND debug info
- `MinSizeRel`, to build with optimizations for code size

CMAKE_INSTALL_PREFIX

This option is to specify the location where the service modules, scripts, configuration files and tools are installed when `make install` is performed. It is set to `/usr/local/nebula` by default.

CMAKE_CXX_COMPILER

Normally, CMake will figure out and locate an applicable C++ compiler for us. But if your compiler installation is not at the standard location, or if you want to use a different one, you have to specify it explicitly as follows:

```
$ cmake -DCMAKE_C_COMPILER=/path/to/gcc/bin/gcc -DCMAKE_CXX_COMPILER=/path/to/gcc/bin/g++ ..
$ cmake -DCMAKE_C_COMPILER=/path/to/clang/bin/clang -DCMAKE_CXX_COMPILER=/path/to/clang/bin/clang++ ..
```

ENABLE_CCACHE

This option is to enable the use of `ccache`, which is for speeding up the compilation during daily development.

By default, **Nebula Graph** will take advantage of `ccache` if it's found. So you don't have to enable it for yourself.

If you want to disable `ccache`, it might be not enough to just turn `ENABLE_CCACHE` off. Since on some platforms, the `ccache` installation hooks up or precedes the compiler. For such a case, you have to set an environment variable `export CCACHE_DISABLE=true`, or add a line `disable=true` to `~/.ccache/ccache.conf`. We may do this for you automatically in future.

Please see the official documentation for more details.

NEBULA_USE_LINKER

This option allows users to use an alternative linker, e.g. `gold`. Options are `bfd`, `lld` and `gold` for now. Among them, `bfd` and `gold` belong to GNU binutils, while `lld` needs to install LLVM / Clang. In addition, you can use this parameter to specify the absolute path of the linker when needed.

NEBULA_THIRDPARTY_ROOT

This option is to explicitly specify the location of the third party.

INSTALLING THIRD PARTY MANUALLY

By default, at the configure(cmake) stage, a prebuilt third party will be downloaded and installed to the current build directory. If you would like to install it into another location for some reason, e.g. to rebuild by removing the whole build directory without downloading the third party again, you could perform the installation manually. Assume you are now at the build directory, run:

```
# To install third party to /opt requires root privilege, you could change it to another location with --prefix.
$ ../third-party/install-third-party.sh --prefix=/opt/vesoft/third-party
```

If the third party is installed to `/opt/vesoft/third-party`, which is by default if no `--prefix` given, the building system of **Nebula Graph** would find it automatically. Otherwise, you need to specify the location with the CMake argument `NEBULA_THIRDPARTY_ROOT` as mentioned above, or set an environment variable to the location and export it. The precedence for **Nebula Graph** to find and choose the third party is:

1. The CMake argument `NEBULA_THIRDPARTY_ROOT`
2. `third-party/install` in the current build directory
3. The `NEBULA_THIRDPARTY_ROOT` environment variable
4. `/opt/vesoft/third-party`

Install an Applicable CMake

For users who don't have a usable CMake installation, we provide a script to automatically download and install one for you. Assuming you are now at the build directory, run:

```
$ ../third-party/install-cmake.sh cmake-install
CMake has been installed to prefix=cmake-install
Run 'source cmake-install/bin/enable-cmake.sh' to make it ready to use.
Run 'source cmake-install/bin/disable-cmake.sh' to disable it.

$ source cmake-install/bin/enable-cmake.sh
$ cmake --version
cmake version 3.15.5
```

Now you have an applicable CMake ready to use. At any time, you could run the command `source cmake-install/bin/disable-cmake.sh` to disable it.

Install an Applicable GCC

For users who don't have a usable GCC installation, we provide a prebuilt GCC and a script to automatically download and install it. Assuming you are now at the build directory, run:

```
# To install GCC to /opt requires root privilege, you could change it to other locations
$ ../third-party/install-gcc.sh --prefix=/opt
GCC-7.5.0 has been installed to /opt/vesoft/toolset/gcc/7.5.0
Performing usability tests
Performing regular C++14 tests...OK
Performing LeakSanitizer tests...OK
Run 'source /opt/vesoft/toolset/gcc/7.5.0/enable' to start using.
Run 'source /opt/vesoft/toolset/gcc/7.5.0/disable' to stop using.

# Please note that the path and specific version might be different from your environment
$ source /opt/vesoft/toolset/gcc/7.5.0/enable
# Only PATH was setup so as not to pollute your library path
# You could run 'export LD_LIBRARY_PATH=/opt/vesoft/toolset/gcc/7.5.0/lib64:$LD_LIBRARY_PATH' if needed

$ g++ --version
g++ (Nebula Graph Build) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
```

Now you have an applicable GCC compiler ready to use. At any time, you could run the command `source /opt/vesoft/toolset/gcc/7.5.0/disable` to disable it.

Building Without Internet Access

For those who don't have an Internet access in the building environment, you have to download the above tools and dependencies manually, including the repo of **Nebula Graph**, GCC compiler, third party and CMake. Then you copy all of these to your building host. Following is a quick guide. Refer to the steps above for more details.

First, in the downloading host:

```
# Please note that although we use command line to illustrate the process, you could perform all the downloading via a browser.

# Download GCC
# For RedHat or CentOS users
$ wget https://oss-cdn.nebula-graph.com.cn/toolset/vesoft-gcc-7.5.0-CentOS-x86_64-glibc-2.12.sh
# For Debian or Ubuntu users
$ wget https://oss-cdn.nebula-graph.com.cn/toolset/vesoft-gcc-7.5.0-Debian-x86_64-glibc-2.13.sh

# Download CMake
$ wget https://cmake.org/files/v3.15/cmake-3.15.5-Linux-x86_64.sh

# Download third party
$ wget https://oss-cdn.nebula-graph.com.cn/third-party/vesoft-third-party-x86_64-libc-2.12-gcc-7.5.0-abi-11.sh
```

Then, copy these packages to the building host, and:

```
# Install GCC
# For RedHat or CentOS users
$ sudo bash vesoft-gcc-7.5.0-CentOS-x86_64-glibc-2.12.sh
# For Debian or Ubuntu users
$ sudo bash vesoft-gcc-7.5.0-Debian-x86_64-glibc-2.13.sh

# Enable the GCC installation
$ source /opt/vesoft/toolset/gcc/7.5.0/enable

# Install CMake
$ sudo bash cmake-3.15.5-Linux-x86_64.sh --skip-license --prefix=/opt/vesoft/toolset/cmake

# Enable CMake by appending its bin directory to PATH
$ export PATH=/opt/vesoft/toolset/cmake:$PATH

# Install third party
$ sudo bash vesoft-third-party-x86_64-libc-2.12-gcc-7.5.0-abi-11.sh
```

Now you are ready to download and build the **Nebula Graph** project.

Uninstalling

Before uninstalling, please stop the services. Use the `make uninstall` command in the `make` directory to uninstall **Nebula Graph**. Please note that the configuration file will not be deleted after uninstalling.

FAQ

ERROR: INVALID ARGUMENT TYPE 'AUTO' TO UNARY EXPRESSION

This error happens when building with Clang 9.0, as shown below:

```
[  5%] Building CXX object src/common/fs/CMakeFiles/fs_obj.dir/FileUtils.cpp.o
In file included from src/common/fs/FileUtils.cpp:8:
In file included from src/common/fs/FileUtils.h:12:
src/common/base/StatusOr.h:57:19: error: invalid argument type 'auto' to unary expression
    static_assert(!is_status_v<T>, "`T' must not be of type `Status'");
                  ^~~~~~~~~~~~~~
src/common/fs/FileUtils.cpp:90:34: note: in instantiation of template class 'nebula::StatusOr<std::__cxx11::basic_string<char> >' requested here
StatusOr<std::string> FileUtils::readLink(const char *path) {
...
```

It is due to a known bug of Clang 9.0 to deal with *auto template variables*, which has not been fixed by Clang 10.0 until now(2020-05-25).

Last update: April 8, 2021

3.1.2 Building With Docker Container

Nebula Graph has provided a docker image with the whole compiling environment vesoft/nebula-dev, which will make it possible to change source code locally, build and debug within the container. Performing the following steps to start quick development:

Pull Image From Docker Hub

```
bash> docker pull vesoft/nebula-dev
```

Run Docker Container

Run docker container and mount your local source code directory into the container working_dir `/home/nebula` with the following command.

```
bash> docker run --rm -ti \
  --security-opt seccomp=unconfined \
  -v /path/to/nebula/directory:/home/nebula \
  -w /home/nebula \
  vesoft/nebula-dev \
  bash
```

Replace `/path/to/nebula/directory` with your **local nebula source code directory**.

Compiling Within the Container

```
docker> mkdir _build && cd _build
docker> cmake ..
docker> make
docker> make install
```

Run Nebula Graph service

Once the preceding installation is completed, you can run **Nebula Graph** service within the container, the default installation directory is `/usr/local/nebula/`.

```
docker> cd /usr/local/nebula
```

Rename config files of **Nebula Graph** service.

```
docker> cp etc/nebula-graphd.conf.default etc/nebula-graphd.conf
docker> cp etc/nebula-metad.conf.default etc/nebula-metad.conf
docker> cp etc/nebula-storaged.conf.default etc/nebula-storaged.conf
```

Start service.

```
docker> ./scripts/nebula.service start all
docker> ./bin/nebula -u root -p nebula --port 3699 --addr="127.0.0.1"
nebula> SHOW HOSTS;
```

Last update: April 8, 2021

3.2 Installation

3.2.1 Nebula Graph Installation with rpm/deb Package

Overview

This guide will walk you through the process of installing **Nebula Graph** with `rpm/deb` packages.

Prerequisites

See Operating Configuration Requirements Doc.

Installing Nebula Graph

To install **Nebula Graph** with a `rpm/deb` package, you must complete the following steps:

1. Download packages.
   - Method one: Download via OSS.

     a. Obtaining the release version information. The URL format is as follows:

     ```
     * Centos 6: https://oss-cdn.nebula-graph.io/package/${release_version}/nebula-${release_version}.el6-5.x86_64.rpm
     * Centos 7: https://oss-cdn.nebula-graph.io/package/${release_version}/nebula-${release_version}.el7-5.x86_64.rpm
     * Ubuntu 1604: https://oss-cdn.nebula-graph.io/package/${release_version}/nebula-${release_version}.ubuntu1604.amd64.deb
     * Ubuntu 1804: https://oss-cdn.nebula-graph.io/package/${release_version}/nebula-${release_version}.ubuntu1804.amd64.deb
     ```

     The `${release_version}` in the link is the release version information. For example, use the follow command to download the 1.2.1 Centos 7 package.

     ```
     $ wget https://oss-cdn.nebula-graph.io/package/1.2.1/nebula-1.2.1.el7-5.x86_64.rpm
     ```

     b. Obtaining the nightly (latest) version. The URL format is as follows:

     ```
     * Centos 6: https://oss-cdn.nebula-graph.io/package/nightly/${date}/nebula-${date}-nightly.el6-5.x86_64.rpm
     * Centos 7: https://oss-cdn.nebula-graph.io/package/nightly/${date}/nebula-${date}-nightly.el7-5.x86_64.rpm
     * Ubuntu 1604: https://oss-cdn.nebula-graph.io/package/nightly/${date}/nebula-${date}-nightly.ubuntu1604.amd64.deb
     * Ubuntu 1804: https://oss-cdn.nebula-graph.io/package/nightly/${date}/nebula-${date}-nightly.ubuntu1804.amd64.deb
     ```

     The `${date}` in the link specifies the date. For example, use the follow command to download the `2020-4-1` `Centos 7.5` package.

     ```
     $ wget https://oss-cdn.nebula-graph.io/package/nightly/2020.04.01/nebula-2020.04.01-nightly.el7-5.x86_64.rpm
     ```

2. Install **Nebula Graph**.
   - For a `rpm` file, install **Nebula Graph** with the following command:

   ```
   sudo rpm -ivh nebula-2019.12.23-nightly.el6-5.x86_64.rpm
   ```

   - For a `deb` file, install **Nebula Graph** with the following command:

   ```
   sudo dpkg -i nebula-2019.12.23-nightly.ubuntu1604.amd64.deb
   ```

   - Install **Nebula Graph** to your customized directory with the following command:

   ```
   rpm -ivh --prefix=${your_dir} nebula-graph-${version}.rpm
   ```

Replace the above file name with your own file name, otherwise, this command might fail. **Nebula Graph** is installed in the `/usr/local/nebula` directory by default.

Starting Nebula Graph Services

See the Start and Stop Nebula Graph Services Doc for details.

Uninstalling

Before uninstalling, please stop the services. If you use rpm to install, use `rpm -qa | grep nebula` command to search for `nebula`, and then pass the result to `rpm -e` to uninstall. For those using deb, you need to uninstall through `dpkg`. Please note that the configuration file will not be deleted after uninstalling.

```
# For rpm
$ rpm -qa|grep nebula
nebula-graph-1.2.1-1.x86_64
$ sudo rpm -e nebula-graph-1.2.1-1.x86_64

# For deb
$ dpkg -l|grep nebula
nebula-graph
$ dpkg -r nebula-graph
```

Last update: April 15, 2021

3.2.2 Start and Stop Nebula Graph Services

Inputting the Following Commands to Start Nebula Graph Services

```
sudo /usr/local/nebula/scripts/nebula.service start all
[INFO] Starting nebula-metad...
[INFO] Done
[INFO] Starting nebula-graphd...
[INFO] Done
[INFO] Starting nebula-storaged...
[INFO] Done
```

Listing Nebula Graph Services

Listing **Nebula Graph** services with the following command:

```
sudo /usr/local/nebula/scripts/nebula.service status all
[INFO] nebula-metad: Running as 9576, Listening on 45500
[INFO] nebula-graphd: Running as 9679, Listening on 3699
[INFO] nebula-storaged: Running as 9812, Listening on 44500
```

Connecting Nebula Graph Service

Connecting **Nebula Graph** service with the following command:

```
sudo /usr/local/nebula/bin/nebula -u <user> -p <password> [--addr=<graphd IP> --port=<graphd port>]

Welcome to Nebula Graph (Version RC4)

nebula> SHOW HOSTS;
```

- -u is the user name, `root` is the default **Nebula Graph** user account
- -p is the password, `nebula` is the default password for account `root`
- --addr is the graphd IP address
- --port is the the graphd server port and the default value is `3699`
- Checking the successfully connected services with command SHOW HOSTS

NOTE: `enable_authorize` is set to `false` by default. You can use any account or password or no account to connect. If enabled, the default user name and password are `root` and `nebula` respectively. See the Built-in Roles Doc.

Stop Nebula Graph Services

Stop **Nebula Graph** services with the following command:

```
sudo /usr/local/nebula/scripts/nebula.service stop all
[INFO] Stopping nebula-metad...
[INFO] Done
[INFO] Stopping nebula-graphd...
[INFO] Done
[INFO] Stopping nebula-storaged...
[INFO] Done
```

Stopping services with `kill -9` might cause data lose. We suggest stopping services in the above way.

Start/Stop Single Nebula Graph Module

Start/stop single module with script nebula.service.

```
sudo /usr/local/nebula/scripts/nebula.service
Usage: ./nebula.service [-v] [-c /path/to/config] <start|stop|restart|status|kill> <metad|graphd|storaged|all>
```

- -v Detailed debugging information of this script
- -c Configuration file path, the default is the `etc/` directory under the installation path (/usr/local/nebula/).

Last update: April 8, 2021

3.2.3 Deploying Cluster

In this document, we will walk you through the process of deploying a **Nebula Graph** cluster. At the same time, we have deployed a cluster with Docker so that you can try it in minutes.

Prerequisites

Before you start deploying the **Nebula Graph** cluster, make sure that you have installed the latest **Nebula Graph** version on each host of your cluster. Installation method reference:

- Installing With rpm
- Building With Source Code

Because **Nebula Graph** have a lot of dependencies, we recommend installing with packages.

In this document, we prepared 3 machines with CentOS 7.5 system, the IPs are as follows:

```
192.168.8.14 # cluster-14
192.168.8.15 # cluster-15
192.168.8.16 # cluster-16
```

Services of Nebula Graph to Be Deployed

In this document, we are going to deploy the following services of **Nebula Graph**:

- 3 replicas of `nebula-metad` service
- 3 replicas of `nebula-storaged` service
- 3 replica of `nebula-graphd` service

```
- cluster-14: metad/storaged/graphd
- cluster-15: metad/storaged/graphd
- cluster-16: metad/storaged/graphd
```

Modifying the Configuration Files

All configuration files of **Nebula Graph** are located in the `/usr/local/nebula/etc` directory. **Nebula Graph** provides three default configurations.

NEBULA-METAD.CONF

When deploying a cluster, you need to modify two parameters in the `nebula-metad.conf` file according to the services deployed on each node: `local_ip` and `meta_server_addrs`. `local_ip` needs to be changed to the node's IP, `meta_server_addrs` needs to be changed to ip:port of the meta service on the cluster. Multiple ip:port pairs need to be separated by commas.

Following is the two configuration on `cluster-14` :

```
# Peers
--meta_server_addrs=192.168.8.14:45500,192.168.8.15:45500,192.168.8.16:45500
# Local ip
--local_ip=192.168.8.14
# Meta daemon listening port
--port=45500
```

NEBULA-GRAPHD.CONF

When deploying a cluster, you need to configure the metad address and port `meta_server_addrs` for the graphd service. Following is part of the configuration on `cluster-14`:

```
# Meta Server Address
--meta_server_addrs=192.168.8.14:45500,192.168.8.15:45500,192.168.8.16:45500
```

NEBULA-STORAGED.CONF

When deploying a cluster, you need to configure the metad address and port `meta_server_addrs` and the local address `local_ip` for the storaged service. Following is part of the configuration on `cluster-14`:

```
# Meta server address
--meta_server_addrs=192.168.8.14:45500,192.168.8.15:45500,192.168.8.16:45500
# Local ip
--local_ip=192.168.8.14
# Storage daemon listening port
--port=44500
```

Starting Your Cluster

Please refer to the Start and stop Nebula Graph service documentation to start your cluster.

Testing Your Cluster

Log in to one host in the cluster and execute the following command:

```
[(none)]> SHOW HOSTS;
=============================================================================================================================================================
| Ip           | Port  | Status | Leader count | Leader distribution                                | Partition distribution                |
=============================================================================================================================================================
| 192.168.8.14 | 44500 | online | 0            | No valid partition                                 | No valid partition                    |
-------------------------------------------------------------------------------------------------------------------------------------------------------------
| 192.168.8.15 | 44500 | online | 3            | toy: 1, test: 1, basketballplayer: 1               | basketballplayer: 1, toy: 1, test: 1 |
-------------------------------------------------------------------------------------------------------------------------------------------------------------
| 192.168.8.16 | 44500 | online | 0            | No valid partition                                 | No valid partition                    |
-------------------------------------------------------------------------------------------------------------------------------------------------------------
| Total        |       |        | 3            | basketballplayer: 1, toy: 1, test: 1               | basketballplayer: 1, test: 1, toy: 1 |
-------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Last update: April 16, 2021

3.2.4 Nebula Graph Installation with Docker

See here.

Last update: April 8, 2021

3.3 Configuration

3.3.1 Operating Configuration Requirements

Production Environment

PRODUCTION ENVIRONMENT DEPLOYMENT METHOD

- 3 metadata service processes `metad`
- At least 3 storage service processes `storaged`
- At least 3 query engine service processes `graphd`

None of the above processes need to monopolize a single machine. For example, a cluster of 5 machines: A, B, C, D, E can be deployed as follows:

- A: metad, storaged, graphd
- B: metad, storaged, graphd
- C: metad, storaged, graphd
- D: storaged, graphd
- E: storaged, graphd

> Do not deploy the same cluster across two IDCs. Each metad process automatically creates and maintains a copy of the metadata, so usually only 3 metad processes are needed. Meanwhile, the number of storaged processes does not affect the copy count of a graph space.

SERVER CONFIGURATION REQUIREMENTS (STANDARD)

Take AWS EC2 c5d.12xlarge as an example:

- CPU: 48 core
- Memory: 96 GB
- Storage: 2 * 900 GB, NVMe SSD
- Linux kernel: 3.9 or higher, check with the command `uname -r`
- glibc: 2.12 or higher, check with the command `ldd --version`

Please refer to the Kernel Configuration Doc for details.

Test Environment

- 1 metadata service process `metad`
- At least 1 storage service process `storaged`
- At least 1 query engine service process `graphd`

For example, a cluster with 3 machines: A, B, C can be deployed as follows:

- A: metad, storaged, graphd
- B: storaged, graphd
- C: storaged, graphd

SERVER CONFIGURATION REQUIREMENTS (MINIMUM)

Take AWS EC2 c5d.xlarge as an example:

- CPU: 4 core
- Memory: 8 GB
- Storage: 100 GB, SSD

Resource Estimation (Three Replicas)

- Storage space (full cluster): number of edges and vertices * average bytes of attributes * 6
- Memory (full cluster): number of edges and vertices * 15 bytes + number of RocksDB instances * (write_buffer_size * max_write_buffer_number) + rocksdb_block_cache * number of the storaged process, where each directory in the --data_path item in the etc/nebula-storaged.conf file corresponds to a RocksDB instance. You can decrease the memory size of bloomfitler by setting the enable_partitioned_index_filter parameter to true. The number of the storaged process usually equals to the number of the machines in the cluster.
- Partitions number of a graph space: number of disks in the cluster * (2 to 10), the better performance of the hard disk, the larger the value.
- Reserve 20% space for memory and hard disk buffer.

About HDD and Gigabit Networks

Nebula Graph is designed for NVMe SSD and 10 Gigabit Network. There is no special adaptation for HDD and gigabit networks. The following are some parameters to be tuned:

- etc/nebula-storage.conf:
  - --raft_rpc_timeout_ms= 5000 to 10000
  - --rocksdb_batch_size= 4096 to 16384
  - --heartbeat_interval_secs = 30 to 60
  - --raft_heartbeat_interval_secs = 30 to 60
- etc/nebula-meta.conf:
  - --heartbeat_interval_secs is the same as etc/nebula-storage.conf
- Spark Writer:

```
rate: {
      timeout: 5000 to 10000
    }
```

- go-importer:
  - batchSize: 10 to 50
  - concurrency: 1 to 10
  - channelBufferSize: 100 to 500
- The partition value is 2 * cluster HDD number

Last update: April 8, 2021

3.3.2 Configuration Persistency and Priority

Configuration Persistency (For Production)

When starting **Nebula Graph** services for the first time, Nebula will read the configuration file from the local (the default path `/usr/local/nebula/etc/`). Then all configuration items (including dynamically changed configuration items) will be **persisted** in the Meta Service. After that, even if restarting **Nebula Graph**, it will only read the configuration from Meta Service.

Getting the Configuration Locally (For Debugging)

In some debugging scenarios, you need to get the configuration from **local** instead of Meta Service. In this case, add `--local_config = true` at the top of the configuration file. You need to restart the services to make the modifications take effect.

Changing Method and Read priority

You can also modify **Nebula Graph** configurations with command lines (`UPDATE CONFIG` syntax) in Nebula console or the environment variables. The read priority rules are as follows:

For a configuration:

- Default configuration precedence: meta > `UPDATE CONFIG` > environment variable > configuration files.
- If set `--local_config` to true, the configuration precedence is: configuration files > meta service > environment variable.

Last update: April 8, 2021

3.3.3 CONFIG Syntax

Introduction to Configuration

Nebula Graph gets configuration from meta by default. If you want to get configuration locally, please add the `--local_config=true` option in the configuration files `metad.conf`, `storaged.conf`, `graphd.conf` (directory is `/home/user/nebula/build/install/etc`) respectively.

NOTE:

- Configuration precedence: meta > console > environment variable > configuration files.
- If set `--local_config` to true, the configuration files take precedence.
- Restart the services after changing the configuration files to take effect.
- Configuration changes in console take effect in real time.

SHOW CONFIGS

```
SHOW CONFIGS [graph|meta|storage]
```

For example:

```
nebula> SHOW CONFIGS meta;
============================================================================================================================
| module | name                                    | type   | mode      | value                                                |
============================================================================================================================
| META   | v                                       | INT64  | IMMUTABLE | 4                                                    |
----------------------------------------------------------------------------------------------------------------------------
| META   | help                                    | BOOL   | IMMUTABLE | False                                                |
----------------------------------------------------------------------------------------------------------------------------
| META   | port                                    | INT64  | IMMUTABLE | 45500                                                |
----------------------------------------------------------------------------------------------------------------------------
```

GET CONFIGS

```
GET CONFIGS [graph|meta|storage :] var
```

For example

```
nebula> GET CONFIGS storage:local_ip;
=====================================================
| module  | name     | type   | mode      | value     |
=====================================================
| STORAGE | local_ip | STRING | IMMUTABLE | 127.0.0.1 |
-----------------------------------------------------
```

```
nebula> GET CONFIGS heartbeat_interval_secs;
=============================================================
| module  | name                    | type  | mode      | value |
=============================================================
| GRAPH   | heartbeat_interval_secs | INT64 | MUTABLE | 10    |
-------------------------------------------------------------
| STORAGE | heartbeat_interval_secs | INT64 | MUTABLE | 10    |
-------------------------------------------------------------
```

UPDATE CONFIGS

```
UPDATE CONFIGS [graph|meta|storage :] var = value
```

The updated CONFIGS will be stored into the `meta service` permanently. If the configuration's mode is `MUTABLE`, the change will take effect immediately. The configurations of some RocksDB parameters take effect after the services are restarted. Expressions are supported in the `UPDATE CONFIGS` command.

For example:

```
nebula> UPDATE CONFIGS storage:heartbeat_interval_secs=1;
nebula> GET CONFIGS storage:heartbeat_interval_secs;
```

```
============================================================
| module  | name                    | type  | mode    | value |
============================================================
| STORAGE | heartbeat_interval_secs | INT64 | MUTABLE | 1     |
------------------------------------------------------------
```

Last update: April 8, 2021

3.3.4 Meta Service Configurations

This document introduces the `metad` configuration file. The default directory is `/usr/local/nebula/etc/`. If you have customized your **Nebula Graph** installation directory, your configuration file path is `$pwd/nebula/etc/`.

- The `*.default` file is used for **debugging** and the **default configuration file** when starting the services
- `*.production` file is the file used for **recommended production**, please remove .production suffix during production

Basic Configurations

Name	Default Value	Descriptions
`daemonize`	true	Run as daemon thread
`pid_file`	"pids/nebula-metad.pid"	File to hold the process ID.

Logging Configurations

Name	Default Value	Descriptions	Dynamic Modification
`log_dir`	logs (i.e. `/usr/local/nebula/logs`)	Directory to metad log. It is recommended to put it on a different hard disk from data_path.	
`minloglevel`	0	The corresponding log levels are INFO(DEBUG), WARNING, ERROR and FATAL. Usually specified as 0 in debug, 1 in production. The minloglevel to 4 prints no logs.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`v`	0	0-4: when minloglevel is set to 0, you can further set the severity level of the debug log. The larger the value, the more detailed the log.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`logbufsecs`	0 (in seconds)	Seconds to buffer the log messages	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.

Networking Configurations

Name	Default Value	Descriptions
`meta_server_addrs`	"127.0.0.1:45500"	A list of meta server IPs. The format is ip1:port1, ip2:port2, ip3:port3. **Configure 3 machines to form a RAFT group in production.**
`port`	45500	RPC daemon listening port. The external port for the Meta service is 45500. The internal `port+1`, namely 45501, is used for the multi-replica interactions.
`reuse_port`	true	Enable Kernel(>3.9) `SO_REUSEPORT` item
`ws_http_port`	11000	HTTP Protocol daemon port. (For internal use)
`ws_h2_port`	11002	HTTP/2 Protocol daemon port. (For internal use)
`ws_ip`	"127.0.0.1"	web service to bind to
`heartbeat_interval_secs`	10 seconds	The same as the parameter in the `nebula-storage.conf` file

NOTE: We recommend you using the actual IP in the `meta_server_addrs` parameter because sometimes `127.0.0.1` will not be parsed correctly.

Storage Configurations

Name	Default Value	Descriptions
`data_path`	data/meta (i.e. /usr/local/nebula/data/meta/)	Directory for cluster metadata persistence

Last update: April 8, 2021

3.3.5 Graph Configurations

This document introduces the `graphd` configuration file. The default directory is `/usr/local/nebula/etc/`. If you have customized your **Nebula Graph** installation directory, your configuration file path is `$pwd/nebula/etc/`.

- The `*.default` file is used for **daily debugging** and the **default configuration file** when starting the services
- `*.production` file is the file used for **recommended production**, please remove .production suffix during production

Basic Configurations

Name	Default Value	Default Value
`daemonize`	true	Run as daemon thread
`pid_file`	"pids/nebula-metad.pid"	File to hold the process ID.

Logging Configurations

Name	Default Value	Descriptions	Dynamic Modification
`log_dir`	logs (i.e. `/usr/local/nebula/logs`)	Directory to graphd log. It is recommended to put it on a different hard disk from data_path.	
`minloglevel`	0	The corresponding log levels are INFO(DEBUG), WARNING, ERROR and FATAL. Usually specified as 0 in debug, 1 in production. The minloglevel to 4 prints no logs.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`v`	0	0-4: when minloglevel is set to 0, you can further set the severity level of the debug log. The larger the value, the more detailed the log.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`logbufsecs`	0 (in seconds)	Seconds to buffer the log messages	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`redirect_stdout`	true	Whether to redirect stdout and stderr to separate files.	
`stdout_log_file`	"stdout.log"	Destination filename of stdout.	
`stderr_log_file`	"stderr.log"	Destination filename of stderr.	
`slow_op_threshhold_ms`	50 (ms)	default threshold for slow operation	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.

For example, change the graphd log level to v=1 with the following command.

```
nebula> UPDATE CONFIGS graph:v=1;
```

Networking Configurations

Name	Default Value	Descriptions	Dynamic Modification
`meta_server_addrs`	"127.0.0.1:45500"	List of meta server addresses, the format looks like ip1:port1, ip2:port2, ip3:port3.	
`port`	3699	RPC daemon's listen port.	
`meta_client_retry_times`	3	meta client retry times	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`heartbeat_interval_secs`	3 (seconds)	Seconds between each heartbeat in meta service.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`client_idle_timeout_secs`	0	Seconds before we close the idle connections, 0 for infinite.	
`session_idle_timeout_secs`	0	Seconds before we expire the idle sessions, 0 for infinite.	
`num_netio_threads`	0	Number of networking threads, 0 for number of physical CPU cores.	
`num_accept_threads`	1	Number of threads to accept incoming connections.	
`num_worker_threads`	0	Number of threads to execute user queries.	
`reuse_port`	true	Whether to turn on the `SO_REUSEPORT` option.	
`listen_backlog`	1024	Backlog of the listen socket.	
`listen_netdev`	"any"	The network device to listen on.	
`ws_http_port`	13000	Port to listen on Graph with HTTP protocol is 13000.	
`ws_h2_port`	13002	Port to listen on Graph with HTTP/2 protocol is 13002.	
`ws_ip`	"127.0.0.1"	IP/Hostname to bind to.	

NOTE: We recommend you using the actual IP in the `meta_server_addrs` parameter because sometimes `127.0.0.1` will not be parsed correctly.

Authorization Configurations

Name	Default Value	Default Value
`enable_authorize`	false	Enable authorize
`auth_type`	password	password: account password; ldap: LDAP; cloud

If you have set `enable_authorize` to true, you can only log in with the root account. For example:

```
/usr/local/nebula/bin/nebula -u root -p nebula --addr=127.0.0.1 --port=3699
```

If you have set `enable_authorize` to false, you can log in without account and password or any account. For example:

```
/usr/local/nebula/bin/nebula --addr=127.0.0.1 --port=3699
```

Last update: April 8, 2021

3.3.6 Storage Configurations

This document introduces the `storaged` configuration file. The default directory is `/usr/local/nebula/etc/`. If you have customized your **Nebula Graph** installation directory, your configuration file path is `$pwd/nebula/etc/`.

- The `*.default` files are used for **daily debugging**. When you start the services, they are the **default configuration files**.
- The `*.production` files are used for the **recommended production**. When they are used for production, the `.production` suffix must be removed.

Basic Configurations

Property	Default Value	Default Value
`daemonize`	true	Run as daemon thread
`pid_file`	"pids/nebula-metad.pid"	File to hold the process ID.

Logging Configurations

Property	Default Value	Descriptions	Dynamic Modification
`log_dir`	logs (i.e. `/usr/local/nebula/logs`)	Directory to storaged log. It is recommended to put it on a different hard disk from data_path.	
`minloglevel`	0	The corresponding log levels are INFO(DEBUG), WARNING, ERROR and FATAL. Usually specified as 0 in debug, 1 in production. The minloglevel to 4 prints no logs.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`v`	0	0-4: when minloglevel is set to 0, you can further set the severity level of the debug log. The larger the value, the more detailed the log.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`slow_op_threshhold_ms`	50 (ms)	default threshhold for slow operation	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.

For example, change the storage log level to v=1 with the following command.

```
nebula> UPDATE CONFIGS storage:v=1;
```

Networking Configurations

Property	Default Value	Descriptions	Dynamic Modification
`meta_server_addrs`	"127.0.0.1:45500"	List of meta server addresses, the format looks like ip1:port1, ip2:port2, ip3:port3.	
`port`	44500	RPC daemon's listen port. The external port for the Storage service is 44500. The internal `port+1`, namely 44501, is used for the multi-replica interactions.	
`reuse_port`	true	Whether to turn on the `SO_REUSEPORT` option.	
`ws_http_port`	12000	HTTP Protocol daemon port. (For internal use)	
`ws_h2_port`	12002	HTTP/2 Protocol daemon port. (For internal use)	
`ws_ip`	"127.0.0.1"	web service to bind to	
`heartbeat_interval_secs`	10 (seconds)	Seconds between each heartbeat. **The same** as the parameter in the `nebula-storage.conf` file.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
`raft_heartbeat_interval_secs`	5 (seconds)	RAFT seconds between each heartbeat.	Modify the configuration file and restart service.
`raft_rpc_timeout_ms`	500 (ms)	RPC timeout for raft client.	Modify the configuration file and restart service.

NOTE: We recommend you using the actual IP in the `meta_server_addrs` parameter because sometimes `127.0.0.1` will not be parsed correctly.

Data persistence setting for storage

Property	Default Value	Descriptions
`data_path`	data/storage (i.e. `/usr/local/nebula/data/storage/`)	The root directory for the local data persistence. If multiple directories exist, use commas to separate the directories. For RocksDB engine, one path one instance.
`auto_remove_invalid_space`	`false`	Whether to remove data from a deleted graph space when restarting the services.

Separate directories when using multiple hard disks. Each directory corresponds to a RocksDB instance for better concurrency. For example:

```
--data_path=/disk1/storage/,/disk2/storage/,/disk3/storage/
```

RocksDB Options

Property	Default Value	Descriptions	Dynamic Modification
rocksdb_batch_size	4096 (B)	Batch Write	
rocksdb_block_cache	1024 (MB)	block cache siez. **Suggest set to 1/3 of the machine memory**	
rocksdb_disable_wal	true	Whether to disable the WAL in RocksDB.	
wal_ttl	14400 (seconds)	RAFT wal time	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.
rocksdb_db_options	{}	jJson string of DBOptions, all keys and values are string.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately. **Overwrite all json**
rocksdb_column_family_options	{}	Json string of ColumnFamilyOptions, all keys and values are string. Details see below.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately. **Overwrite all json**
rocksdb_block_based_table_options	{}	Json string of BlockBasedTableOptions, all keys and values are string. Details see below.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately. **Overwrite all json**

ROCKSDB_DB_OPTIONS

```
max_total_wal_size
delete_obsolete_files_period_micros
max_background_jobs
stats_dump_period_sec
compaction_readahead_size
writable_file_max_buffer_size
bytes_per_sync
wal_bytes_per_sync
delayed_write_rate
avoid_flush_during_shutdown
max_open_files
stats_persist_period_sec
stats_history_buffer_size
strict_bytes_per_sync
enable_rocksdb_prefix_filtering
enable_rocksdb_whole_key_filtering
rocksdb_filtering_prefix_length
num_compaction_threads
rate_limit
```

The parameters above can either be dynamically modified by the `UPDATE CONFIGS` syntax, or written in the local configuration file. Please refer to the RocksDB manual for specific functions and whether restarting is needed.

ROCKSDB_COLUMN_FAMILY_OPTIONS

```
write_buffer_size
max_write_buffer_number
level0_file_num_compaction_trigger
level0_slowdown_writes_trigger
level0_stop_writes_trigger
target_file_size_base
target_file_size_multiplier
max_bytes_for_level_base
```

```
max_bytes_for_level_multiplier
disable_auto_compactions       -- Compact automatically when writing data is stopped, default value is false. Dynamic modification takes effect immediately.
```

The preceding parameters can either be dynamically modified by UPDATE CONFIGS syntax, or written in the local configuration file. Please refer to the RocksDB manual for specific functions and whether restarting is needed.

The preceding parameters can be set via the command line as follows:

```
nebula> UPDATE CONFIGS storage:rocksdb_column_family_options = \
        { disable_auto_compactions = false, level0_file_num_compaction_trigger = 10 };
        -- The command overwrites rocksdb_column_family_options. Please note whether other sub-items will be overwritten

nebula> UPDATE CONFIGS storage:rocksdb_db_options  = \
        { max_subcompactions = 10, max_background_jobs = 10};
nebula> UPDATE CONFIGS storage:max_edge_returned_per_vertex = 10; -- The parameter is explained below
```

We recommend the following configuration:

```
    rocksdb_db_options = {"stats_dump_period_sec":"200", "write_thread_max_yield_usec":"600"}
    rocksdb_column_family_options = {"max_write_buffer_number":"4", "min_write_buffer_number_to_merge":"2", "max_write_buffer_number_to_maintain":"1"}
    rocksdb_block_based_table_options = {"block_restart_interval":"2"}
```

Description on Super-Large Vertices

For super vertex with a large number of edges, currently there are two truncation strategies:

1. Truncate directly. Set the `enable_reservoir_sampling` parameter to `false`. A certain number of edges specified in the `Max_edge_returned_per_vertex` parameter are truncated by default.
2. Truncate with the reservoir sampling algorithm. Based on the algorithm, a certain number of edges specified in the `Max_edge_returned_per_vertex` parameter are truncated with equal probability from the total n edges. Equal probability sampling is useful in some business scenarios. However, the performance is effected compared to direct truncation due to the probability calculation.

For example:

```
nebula> UPDATE CONFIGS storage:enable_reservoir_sampling = false;
```

TRUNCATING DIRECTLY

Property	Default Value	Descriptions	Dynamic Modification
max_edge_returned_per_vertex	2147483647	The max returned edges of each super-large vertex. The excess edges are truncated and not returned.	
Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.			

RESERVOIR SAMPLING TRUNCATION

Property	Default Value	Descriptions	Dynamic Modification
enable_reservoir_sampling	false	Truncated with equal probability from the total n edges.	Modified with UPDATE CONFIGS syntax. The modification takes effect immediately.

Storage Configuration When You Have a Lot of Data

If you have a lot of data (in the RocksDB directory) and your memory is tight, we recommend you setting the `enable_partitioned_index_filter` parameter in the storage configuration to `true`. For example, 100 vertices + 100 edges occupies 300 keys. Each key is 10bit or 3k. Then you can calculate your own data storage.

Last update: April 8, 2021

3.3.7 Console Configurations

Name	Default Value	Description
`addr`	"127.0.0.1"	Graph daemon IP address.
`port`	0	Graph daemon listening port.
`u`	""	Username used to authenticate.
`p`	""	Password used to authenticate.
`enable_history`	false	Whether to force saving the command history.
`server_conn_timeout_ms`	1000	Connection timeout in milliseconds.

Last update: April 8, 2021

3.3.8 Kernel Configuration Reference

This section contains a reference of **Nebula Graph** system configuration settings.

ulimit

ULIMIT -C

It limits the size of core dumps. The recommended setting is *unlimited*, i.e. `ulimit -c unlimited`.

ULIMIT -N

It limits a user by number of opened files (file descriptors). The recommended setting is greater than 100,000. For example, `ulimit -n 130000`.

Memory

VM.SWAPPINESS

It controls the *relative weight* given to swapping out of runtime memory, as opposed to dropping memory pages from the system page cache. The lower the value, the less swapping is used and the more memory pages are kept in physical memory. The recommended setting is 0. Note that 0 does not mean that there is no swapping out.

VM.MIN_FREE_KBYTES

It is used to force the Linux VM to keep a minimum number of kilobytes free. If the system physical memory is large, it is recommended to increase this value (for example, if your physical memory 128GB, set it to 5GB). Otherwise, when there is not enough memory, the system cannot apply for large continuous physical memory.

VM.MAX_MAP_COUNT

It contains the maximum number of vma (virtual memory area) a process may have. The default value is 65530. If the memory application fails when the memory consumption is large, you can increase it appropriately.

VM.OVERCOMMIT_MEMORY

This value contains a flag that enables memory overcommitment. It is recommended to set it to the default value 0 or 1. Do not set it to 2.

VM.DIRTY_*

These values contain the the aggressiveness of the dirty page cache that controls the system. For write-intensive scenarios, you can make adjustments based on your needs (throughput priority or delay priority). It is recommended to use the system default.

TRANSPARENT HUGE PAGE

Transparent Huge Pages (THP) must be disabled for better delay performance. The options are `/sys/kernel/mm/transparent_hugepage/enabled` and `/sys/kernel/mm/transparent_hugepage/defrag`. For example:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
swapoff -a && swapon -a
```

Networking

NET.IPV4.TCP_SLOW_START_AFTER_IDLE

If set to the default value 1, it will time out the congestion window after an idle period. It is recommended to set to 0, especially for long fat links (high latency and large bandwidth).

NET.CORE.SOMAXCONN

This option, together with the backlog parameter called by the listen system, specifies the upper limit on how many connections the kernel will accept. For scenarios where a large number of burst connections are established, it is recommended to set it above 1024 (the default value is 128).

NET.IPV4.TCP_MAX_SYN_BACKLOG

Maximum number of remembered connection requests. The setting rule is the same as `net.core.somaxconn`.

NET.CORE.NETDEV_MAX_BACKLOG

It determines the maximum number of packets. It is suggested raising this value to over 10,000 for high-throughput scenarios, especially 10G network adapters. The default value is 1,000.

NET.IPV4.TCP_KEEPALIVE_*

Parameters to enable the TCP keep alive feature. For applications that use a 4-layer transparent load balancer, if the idle connection is unexpectedly disconnected, you can decrease `tcp_keepalive_time` and `tcp_keepalive_intvl`.

NET.IPV4.TCP_RMEM/WMEM

They represent the minimum, default and maximum size of the TCP socket receive buffer. For long fat links, it is recommended to increase the default value to bandwidth * RTT.

SCHEDULER

For SSD devices, it is recommended to set `/sys/block/DEV_NAME/queue/scheduler` to noop or none.

Other Parameters

KERNEL.CORE_PATTERN

It is recommended to set it to core and set `kernel.core_uses_pid` to 1.

Parameter Usage Guide

SYSCTL

- `sysctl conf_name` checks the current parameter value
- `sysctl -w conf_name=value` modifies the parameter value and takes effect immediately
- `sysctl -p` loads parameter values from relevant configuration files

INTRODUCTION TO ULIMIT

`ulimit` provides control over the resources available to the shell and to processes started by it, on systems that allow such control. Please note that:

- Changes made by the `ulimit` command are only valid for the current session (and child processes).
- `ulimit` cannot adjust the (soft) threshold of a resource to be greater than the current hard value.
- Ordinary users cannot adjust the hard threshold (even using `sudo`) through this command.
- To modify on the system level, or adjust the hard threshold, you need to edit the `/etc/security/limits.conf` file. But this method needs to log in again to take effect.

PRLIMIT

`prlimit` tries to retrieve and/or modify the limits to a specified process. Together with the `sudo` command, the hard threshold can be modified. For example, `prlimit --nofile = 130000 --pid = $$` can adjust the maximum number of open files allowed by the current process to 14000 and take effect immediately. Please note that this command is only available in RedHat 7u or later OS versions.

Last update: April 8, 2021

3.3.9 Logs

Nebula Graph uses glog to print logs, uses gflag to control the severity level of the log, and provides an HTTP interface to dynamically change the log level at runtime to facilitate tracking.

Log Directory

The default log directory is `/usr/local/nebula/logs/`.

NOTE: If you deleted the log directory during runtime, the runtime log would not continue to be printed. However, this operation will not affect the services. Restart the services to recover the logs.

Parameter Description

TWO MOST COMMONLY USED FLAGS IN GLOG

- minloglevel: The scale of minloglevel is 0-4. The numbers of severity levels INFO(DEBUG), WARNING, ERROR, and FATAL are 0, 1, 2, and 3, respectively. Usually specified as 0 for debug, 1 for production. If you set the minloglevel to 4, no logs are printed.
- v: The scale of v is 0-3. When the value is set to 0, you can further set the severity level of the debug log. The greater the value is, the more detailed the log is.

CONFIGURATION FILES

The default severity level for the metad, graphd, and storaged logs can be found in the configuration files (usually in `/usr/local/nebula/etc/`).

Check and Change the Severity Levels Dynamically

Check all the flag values (log values included) of the current gflags with the following command. `curl` is only available when the `local_config` parameter is set to `true`.

```
> curl ${ws_ip}:${ws_port}/get_flags
```

In the command:

- `ws_ip` is the IP address for the HTTP service, which can be found in the configuration files above. The default value is 127.0.0.1.
- `ws_port` is the port for the HTTP service, the default values for `metad`, `storaged`, and `graphd` are 11000, 12000, and 13000, respectively.

For example, check the minloglevel for the `storaged` service:

```
> curl 127.0.0.1:12000/get_flags | grep minloglevel
```

You can also change the logs' severity level to **the most detailed** with the following command.

```
> curl "http://127.0.0.1:12000/set_flags?flag=v&value=4"
> curl "http://127.0.0.1:12000/set_flags?flag=minloglevel&value=0"
```

In the Nebula console, check the severity minloglevel of graphd and set it to **the most detailed** with the following commands.

```
nebula> GET CONFIGS graph:minloglevel;
nebula> UPDATE CONFIGS graph:minloglevel=0;
```

To change the severity of the storage log, replace the `graph` in the preceding command with `storage`.

NOTE: **Nebula Graph** only supports modifying the graph and storage log severity by using console. And the severity level of meta logs can only be modified with the `curl` command.

Close all logs print (FATAL only) with the following command.

```
> curl "http://127.0.0.1:12000/set_flags?flag=minloglevel&value=4"
```

Last update: April 8, 2021

3.4 Account Management Statement

3.4.1 Alter User Syntax

```
ALTER USER <user_name> WITH PASSWORD <password>
```

The `ALTER USER` statement modifies **Nebula Graph** user accounts. `ALTER USER` requires the global `CREATE USER` privilege. An error occurs if you try to modify a user that does not exist. `ALTER` does not require password verification.

Last update: April 8, 2021

3.4.2 Authentication

Whenever a client connects to **Nebula Graph**, a session is created. The session stores various contextual information about the connection. Each session is always associated with a single user.

Authentication is the process of mapping this session to a specific user. Once the session is mapped to a user, a set of permissions can be associated with it, using authorization.

Nebula Graph supports two authentication methods, explained in detail below - local and LDAP.

Local authentication

The local database stores usernames, encrypted passwords, local user settings and remote LDAP user settings. When a user tries to access the database, they will be met with a security challenge.

To enable the local authentication, follow these steps:

1. Set the `--enable_authorize` property in the `nebula-graphd.conf` configuration file (the directory is `/usr/local/nebula/etc/` by default) to `true`.
2. Save your modification in step one and close the `nebula-graphd.conf` configuration file.
3. Restart the Nebula Graph services.

LDAP authentication

Lightweight Directory Access Protocol (LDAP) is a lightweight client-server protocol for accessing directory services. Users stored inside LDAP take precedence over the local database users. For example, if both providers have a user called “Amber”, the settings and roles for this user will be sourced from LDAP.

Unlike local authentication, besides enabling the `--enable_authorize` parameter, LDAP needs to be configured in the `nebula-graphd.conf` file (the directory is `/usr/local/nebula/etc/` by default). Refer to the Integrating LDAP Document for details.

LDAP PARAMETERS

Parameter	Type	Default Value	Description
ldap_server	string	""	A list of ldap server addresses. Multiple addresses are separated with commas.
ldap_port	INT32	Ldap server port. If no port is specified, the default port will be used.	
ldap_scheme	string	"ldap"	Only supports ldap.
ldap_tls	bool	false	Enable/disable the TLS encryption between graphd and the LDAP server.
ldap_suffix	string	""	Specifies the root suffix (naming context) to use for all LDAP operations.
ldap_basedn	string	""	The LDAP distinguished name (DN) of the search base.
ldap_binddn	string	""	The LDAP user who is allowed to search the base DN.
ldap_bindpasswd	string	""	The password of the user who is mentioned in the bind DN.
ldap_searchattribute	string	""	An array of the required attributes.
ldap_searchfilter	string	""	Specifies a search filter by defining what to search for. It is more flexible than the searchattribut.

FAQ

ERROR INFORMATION: AUTHENTICATION FAILS, INVALID DATA LENGTH

Authentication fails because you had not enable the authentication. Follow the preceding steps to enable the authentication.

Last update: April 8, 2021

3.4.3 Built-in Roles

Nebula Graph provides the following roles:

- God
  - The initial root user (similar to the Root in Linux and Administrator in Windows).
  - All the operation access.
  - A cluster can only have one God. God manages all the spaces in the cluster.
  - When a cluster is initialized, a default GOD account named root is created.
  - The God role is automatically initialized by meta and cannot be granted by users.
- Admin
  - The administration user.
  - Read/write access to both the schema and data limited to its authorized space.
  - Authorization access to users limited to its authorized space.
- DBA
  - Read/write access to both the schema and data limited to its authorized space.
  - No authorization access to users.
- User
  - Read/write access to data limited to its authorized space.
  - Read-only access to the schema limited to its authorized space.
- Guest
  - Read-only access to both the schema and data limited to its authorized space.

If the authorization is enabled, the default user name and password are `root` and `nebula` respectively, and the user name is immutable. Set the `enable_authorize` parameter in the `/usr/local/nebula/etc/nebula-graphd.conf` file to `true` to enable the authorization.

A user who has no assigned roles will not have any accesses to the space. A user can only have one assigned role in the same space. A user can have different roles in different spaces.

The set of executor prescribed by each role are described below.

Divided by operation permissions.

OPERATION	STATEMENTS
Read space	Use, DescribeSpace
Write space	CreateSpace, DropSpace, CreateSnapshot, DropSnapshot, Balance, Admin, Config, Ingest, Download
Read schema	DescribeTag, DescribeEdge, DescribeTagIndex, DescribeEdgeIndex
Write schema	CreateTag, AlterTag, CreateEdge, AlterEdge, DropTag, DropEdge, CreateTagIndex, CreateEdgeIndex, DropTagIndex, DropEdgeIndex
Write user	CreateUser, DropUser, AlterUser
Write role	Grant, Revoke
Read data	Go, Set, Pipe, Match, Assignment, Lookup, Yield, OrderBy, FetchVertices, Find, FetchEdges, FindPath, Limit, GroupBy, Return
Write data	BuildTagIndex, BuildEdgeIndex, InsertVertex, UpdateVertex, InsertEdge, UpdateEdge, DeleteVertex, DeleteEdges
Special operation	Show, ChangePassword

Divided by operations.

OPERATION	GOD	ADMIN	DBA	USER	GUEST
Read space	Y	Y	Y	Y	Y
Write space	Y				
Read schema	Y	Y	Y	Y	Y
Write schema	Y	Y	Y		
Write user	Y				
Write role	Y	Y			
Read data	Y	Y	Y	Y	Y
Write data	Y	Y	Y	Y	
Special operation	Y	Y	Y	Y	Y
Show Jobs	Y	Y	Y	Y	Y

NOTE: Pay attention to the special operation here. The returned results vary based on the account role. For example, each role has the access to the `SHOW SPACE` statement, but the returned results vary based on the account authentication. Only the `GOD` user has the authority to run the `SHOW USERS` and the `SHOW SNAPSHOTS` statements.

Last update: April 8, 2021

3.4.4 CHANGE PASSWORD Syntax

```
CHANGE PASSWORD <user_name> FROM <old_psw> TO <new-psw>
```

The `CHANGE PASSWORD` statement changes a password to a **Nebula Graph** user account. The old password is required in addition to the new one.

Last update: April 8, 2021

3.4.5 Create User Syntax

```
CREATE USER [IF NOT EXISTS] <user_name> [WITH PASSWORD <password>]
```

The `CREATE USER` statement creates new **Nebula Graph** accounts. To use `CREATE USER`, you must have the global `CREATE USER` privilege. By default, an error occurs if you try to create a user that already exists. If the `IF NOT EXISTS` clause is given, the statement produces a warning for each named user that already exists, rather than an error.

For example:

```
nebula> CREATE USER user1 WITH PASSWORD 'nebula';
```

Last update: April 8, 2021

3.4.6 Drop User Syntax

```
DROP USER [IF EXISTS] <user_name>
```

Only `God` and `Admin` users have the DROP privilege for the sentence.

`DROP USER` does not automatically close any already opened client session.

Last update: April 8, 2021

3.4.7 Grant Role Syntax

```
GRANT ROLE <role_type> ON <space> TO <user>
```

The `GRANT` statement assigns role to **Nebula Graph** user account. To use `GRANT`, you must have the `GRANT` privilege.

Currently, there are five roles in **Nebula Graph**: `GOD`, `ADMIN`, `DBA`, `USER` and `GUEST`.

Normally, first use `CREATE USER` to create an account then use `GRANT` to define its privileges (assume you have the `CREATE` and `GRANT` privilege). Each role and user account to be granted must exist, or errors will occur.

The `<space>` clause must be specified as an existed graph space or an error will occur.

Last update: April 8, 2021

3.4.8 Integrating LDAP

This document describes how to connect **Nebula Graph** to a LDAP server for authentication (only available with the Enterprise Edition).

About LDAP Integration

LDAP integration allows you to share user identity information and passwords defined in LDAP with **Nebula Graph**.

Installing LDAP Plugin

1. Build the LDAP server and insert the corresponding record.

   For example, insert the username `test2` with the corresponding password `passwdtest2`. Then check the user by the following command:

   ```
   ldapsearch -x -b 'uid=test2,ou=it,dc=sys,dc=com'
   ```

2. Put the `auth_ldap.so` file in the shared directory of the installation path.
3. Create a shadow account to install the `auth_ldap` plugin.

   Login to **Nebula Graph** as root with password `nebula` and the `auth_type` is `password`:

   ```
   ./bin/nebula -u root  -p nebula   --port 3699 --addr="127.0.0.1"
   ```

   Create a shadow account:

   ```
   # You need to authorize the shadow account test2 first
   nebula> CREATE USER test2 WITH PASSWORD "";
   ```

   Install the `auth_ldap` plugin:

   ```
   nebula> INSTALL PLUGIN auth_ldap SONMAE "auth_ldap.so";
   ```

   Check whether the plugin is installed successfully:

   ```
   nebula> SHOW PLUGINS;
   ```

Uninstalling LDAP Plugin

1. Login to **Nebula Graph** as root with password `nebula` and the `auth_type` is `password`:

   ```
   ./bin/nebula -u root  -p nebula   --port 3699 --addr="127.0.0.1"
   ```

2. Run the following command to uninstall the `auth_ldap` plugin:

   ```
   nebula> UNINSTALL PLUGIN auth_ldap;
   ```

Enabling LDAP Authentication in Nebula Graph

LDAP authentication in **Nebula Graph** is disabled by default. To enable LDAP authentication in **Nebula Graph**:

ENABLING THE AUTHORIZATION

First, enable the authorization. Open the `nebula-graphd.conf` file (the directory is `/usr/local/nebula/etc/` by default), and locate the `--enable_authorize` property. Change the value of the property to true:

```
########## Authorization ##########
# Enable authorization
--enable_authorize=true
```

CONFIGURING NEBULA GRAPH

Then configure **Nebula Graph** to use LDAP. There are two LDAP methods that you use to authenticate your **Nebula Graph** services against an LDAP server.

- **Simple bind authentication**. Open the `nebula-graphd.conf` file (the directory is `/usr/local/nebula/etc/` by default), and locate the `Authentication` section. Change the value of the `auth_type` to `ldap` and add the following properties:

```
########## Authentication ##########
# User login authentication type, password for nebula authentication, ldap for ldap authentication, cloud for cloud authentication
--auth_type=ldap
--ldap_server=127.0.0.1
--ldap_port=389
--ldap_scheme=ldap
--ldap_prefix=uid=
--ldap_suffix=,ou=it,dc=sys,dc=com
```

- **Search and bind authentication**. Open the `nebula-graphd.conf` file (the directory is `/usr/local/nebula/etc/` by default), and locate the `Authentication` section. Change the value of the `auth_type` to `ldap` and add the following properties:

```
########## Authentication ##########
# User login authentication type, password for nebula authentication, ldap for ldap authentication, cloud for cloud authentication
--auth_type=ldap
--ldap_server=127.0.0.1
--ldap_port=389
--ldap_scheme=ldap
--ldap_prefix=uid=
--ldap_suffix=,ou=it,dc=sys,dc=com
```

RESTART SERVICES

Save and close the file. Restart the services:

```
/usr/local/nebula/scripts/nebula.service restart all
```

Disabling LDAP Authentication in Nebula Graph

You can disable LDAP authentication in **Nebula Graph** by setting the `--enable_authorize` parameter to `false` in the `nebula-graphd.conf` file and restarting the services.

Connecting to Nebula Graph Through LDAP Authentication

Once configuration completes, you can connect to **Nebula Graph** through LDAP authentication with the following command:

```
./bin/nebula -u test2 -p passwdtest2 --port 3699 --addr="127.0.0.1"
```

Last update: April 8, 2021

3.4.9 Revoke Syntax

```
REVOKE ROLE <role_type> ON <space> FROM <user>
```

The `REVOKE` statement removes access privileges from **Nebula Graph** user accounts. To use `REVOKE`, you must have the `REVOKE` privilege.

Currently, there are five roles in **Nebula Graph**: `GOD`, `ADMIN`, `DBA`, `USER` and `GUEST`.

User accounts and roles are to be revoked must exist, or errors will occur.

The `<space>` clause must be specified as an existed graph space or an error will occur.

Last update: April 8, 2021

3.5 Batch Data Management

3.5.1 Data Import

Import csv File

See vesoft-inc/nebula-importer.

Last update: April 8, 2021

Spark Writer

OVERVIEW

Spark Writer is a Spark-based distributed data importer for Nebula Graph. It converts data from multiple data sources into vertices and edges for graphs and batch imports data into the graph database. Currently, the supported data sources are:

- HDFS, including Parquet, JSON, ORC and CSV
- HIVE

Spark Writer supports concurrent importing multiple tags and edges, and configuring different data sources for different tags and edges.

PREREQUISITES

NOTE: To use **Nebula Graph Spark Writer**, please make sure you have:

- Spark 2.0 or above
- Hive 2.3 or above
- Hadoop 2.0 or above

GET SPARK WRITER

From Source Code

```
git clone https://github.com/vesoft-inc/nebula.git
cd nebula/src/tools/spark-sstfile-generator
mvn compile package
```

Or you can download from OSS.

Download From Cloud Storage OSS

```
wget https://oss-cdn.nebula-graph.io/jar-packages/sst.generator-1.2.1-beta.jar
```

USER GUIDE

This section includes the following steps:

1. Create a graph space and its schema in Nebula Graph
2. Write data files
3. Write input source mapping file
4. Import data

Create Graph Space

Please refer to the example graph in Quick Start.

NOTE: Please create a space and define the schema in Nebula Graph first, then use this tool to import data to Nebula Graph.

Example Data

Vertices

A vertex data file consists of multiple rows. Each line in the file represents a vertex and its properties. In general, the first column is the ID of the vertex. This ID column is specified in the mapping file. The other columns are the properties of the vertex. Consider the following example in JSON format.

- **Player** data

```
{"id":100,"name":"Tim Duncan","age":42}
{"id":101,"name":"Tony Parker","age":36}
{"id":102,"name":"LaMarcus Aldridge","age":33}
```

Edges

An edge data file consists of multiple rows. Each line in the file represents a point and its properties. In general, the first column is the ID of the source vertex, the second column is the ID of the destination vertex. These ID columns are specified in the mapping file. Other columns are the properties of the edge. Consider the following example in JSON format.

Take edge ***follow*** as example:

- Edge without rank

```
{"source":100,"target":101,"likeness":95}
{"source":101,"target":100,"likeness":95}
{"source":101,"target":102,"likeness":90}
```

- Edge with rank

```
{"source":100,"target":101,"likeness":95,"ranking":2}
{"source":101,"target":100,"likeness":95,"ranking":1}
{"source":101,"target":102,"likeness":90,"ranking":3}
```

Spatial Data Geo

Spark Writer supports importing Geo data. Geo data contains **latitude** and **longitude**, and the data type is double.

```
{"latitude":30.2822095,"longitude":120.0298785,"target":0,"dp_poi_name":"0"}
{"latitude":30.2813834,"longitude":120.0208692,"target":1,"dp_poi_name":"1"}
{"latitude":30.2807347,"longitude":120.0181162,"target":2,"dp_poi_name":"2"}
{"latitude":30.2812694,"longitude":120.0164896,"target":3,"dp_poi_name":"3"}
```

Data Source Files

The currently supported data sources by Spark Writer are:

- HDFS
- HIVE

HDFS Files

HDFS supports the following file formats:

- Parquet
- JSON
- CSV
- ORC

Player data in Parquet format:

```
+-------+---+---------+
|age| id|        name|
+-------+---+---------+
| 42|100| Tim Duncan |
| 36|101| Tony Parker|
+-------+---+---------+
```

In JSON:

```
{"id":100,"name":"Tim Duncan","age":42}
{"id":101,"name":"Tony Parker","age":36}
```

In CSV:

```
age,id,name
42,100,Tim Duncan
36,101,Tony Parker
```

Database

Spark Writer supports database as the data source, and only HIVE is available now.

Player format as follows:

col_name	data_type	comment
id	int	
name	string	
age	int	

Write Configuration Files

The configuration files consist of the Spark field, the Nebula Graph field, the tags mapping field, and the edges mapping field. The Spark related parameters are configured in the Spark field. The username and password information for Nebula Graph are configured in the Nebula Graph field. Basic data source information for each tag or edge is described in the tag/edge mapping field. The tag/edge mapping field corresponds to multiple tag/edge inputting sources. Different tag/edge can come from different data sources.

Example of a mapping file for the input source:

```
{
  # Spark related configurations.
  # See also: http://spark.apache.org/docs/latest/configuration.html
  spark: {
    app: {
      name: Spark Writer
    }

    driver: {
      cores: 1
      maxResultSize: 1G
    }

    cores {
      max: 16
    }
  }

  # Nebula Graph related configurations.
  nebula: {
    # Query engine IP list
    addresses: ["127.0.0.1:3699"]

    # Username and password to connect to Nebula Graph service
    user: user
    pswd: password

    # Graph space name for Nebula Graph
    space: test

    # The thrift connection timeout and retry times.
    # If no configurations are set, the default values are 3000 and 3 respectively.
    connection {
      timeout: 3000
      retry: 3
    }

  # The nGQL execution retry times.
  # If no configuration is set, the default value is 3.
    execution {
      retry: 3
    }
  }

  # Processing tags
  tags: [

    # Loading tag from HDFS and the data type is parquet.
    # The tag's name is tag_name_0.
    # field_0, field_1 and field_2 from HDFS's Parquet file are written into tag_name_0
    # and the vertex column is vertex_key_field.
    {
      name: tag_name_0
      type: parquet
      path: hdfs_path
      fields: {
        field_0: nebula_field_0,
        field_1: nebula_field_1,
        field_2: nebula_field_2
      }
      vertex: vertex_key_field
      batch : 16
    }

    # Similar to the preceding section.
```

```
    # Loaded from Hive. The execution command $ {EXEC} is the dataset.
    {
      name: tag_name_1
      type: hive
      exec: "select hive_field_0, hive_field_1, hive_field_2 from database.table"
      fields: {
        hive_field_0: nebula_field_0,
        hive_field_1: nebula_field_1,
        hive_field_2: nebula_field_2
      }
      vertex: vertex_id_field
    }
  ]

  # Processing edges
  edges: [
    # Loading edge from HDFS and data type is JSON.
    # The edge's name is edge_name_0.
    # field_0, field_1 and field_2 from HDFS's JSON file are written into edge_name_0
    # The source column is source_field, target column is target_field and ranking column is ranking_field.
    {
      name: edge_name_0
      type: json
      path: hdfs_path
      fields: {
        field_0: nebula_field_0,
        field_1: nebula_field_1,
        field_2: nebula_field_2
      }
      source:  source_field
      target:  target_field
      ranking: ranking_field
    }

    # Loading from Hive will execute command ${exec} as data set.
    # Ranking is optional.
    {
      name: edge_name_1
      type: hive
      exec: "select hive_field_0, hive_field_1, hive_field_2 from database.table"
      fields: {
        hive_field_0: nebula_field_0,
        hive_field_1: nebula_field_1,
        hive_field_2: nebula_field_2
      }
      source:  source_id_field
      target:  target_id_field
    }
  ]
}
```

Spark Properties

The following table gives some example properties, all of which can be found in Spark Available Properties.

Field	Default	Required	Description
spark.app.name	Spark Writer	No	The name of your application
spark.driver.cores	1	No	Number of cores to use for the driver process, only in cluster mode.
spark.driver.maxResultSize	1G	No	Limit of total size of serialized results of all partitions for each Spark action (e.g. collect) in bytes. It must be at least 1M, or 0 for unlimited.
spark.cores.max	(not set)	No	When running on a standalone deploy cluster or a Mesos cluster in "coarse-grained" sharing mode, the maximum amount of CPU cores to request for the application from across the cluster (not from each machine). If not set, the default will be `spark.deploy.defaultCores` on Spark's standalone cluster manager, or infinite (all available cores) on Mesos.

Nebula Graph Configuration

Field	Default Value	Required	Description
nebula.addresses	/	yes	query engine IP list, separated with comma
nebula.user	/	yes	user name, the default value is user
nebula.pswd	/	yes	password, the default `user` password is `password`
nebula.space	/	yes	space to import data, the space name is test in this document
nebula.connection.timeout	3000	no	Thrift timeout
nebula.connection.retry	3	no	Thrift retry times
nebula.execution.retry	3	no	nGQL execution retry times

Mapping of Tags and Edges

The options for tag and edge mapping are very similar. The following describes the same options first, and then introduces the unique options of `tag mapping` and `edge mapping`.

- **Same Options**
  - `type` is a case insensitive required field that specifies data type in the context, and currently supports Parquet, JSON, ORC and CSV
  - `path` is applied to HDFS data source and specifies the absolute path of HDFS file or directory. It is a required field when the type is HDFS
  - `exec` is applied to Hive data source. It is a required filed when the query type is HIVE
  - `fields` is a required filed that maps the columns of the data source to properties of tag / edge
- **unique options for tag mapping**
  - `vertex` is a required field that specifies a column as the vertex ID column
- **unique options for edge mapping**
  - `source` is a required field that specifies a column in the input source as the **source vertex** ID column
  - `target` is a required field that specifies a column as the **destination vertex** ID column
  - `ranking` is an optional field that specifies a column as the edge ranking column when the inserted edge has a ranking value

Data Source Mapping

- **HDFS Parquet Files**
  - `type` specifies the input source type. When it is parquet, it is a case insensitive required field
  - `path` specifies the HDFS file directory. It is a required field that must be the absolute directory
- **HDFS JSON Files**
  - `type` specifies the type of the input source. When it is JSON, it is a case insensitive required field
  - `path` specifies the HDFS file directory. It is a required field that must be absolute directory
- **HIVE ORC Files**
  - `type` specifies the input source type. When it is ORC, it is a case insensitive required field
  - `path` specifies the HDFS file directory. It is a required field that must be the absolute directory
- **HIVE CSV Files**
  - `type` specifies the input source type. When it is CSV, it is a case insensitive required field
  - `path` specifies the HDFS file directory. It is a required field that must be the absolute directory
- **HIVE**
  - `type` specifies the input source type. When it is HIVE, it is a case insensitive required field
  - `exec` is a required field that specifies the HIVE executed query

Import Data

Import data with the following command:

```
bin/spark-submit \
 --class com.vesoft.nebula.tools.generator.v2.SparkClientGenerator \
 --master ${MASTER-URL} \
 ${SPARK_WRITER_JAR_PACKAGE} -c conf/test.conf -h -d
```

Parameter descriptions:

Abbreviation	Required	Default	Description	Example
--class	yes	/	Specify the program's primary class	
--master	yes	/	Specify spark cluster master url. Refer to master urls for detail	e.g. `spark://23.195.26.187:7077`
-c / --config	yes	/	The configuration file path in the context	
-h / --hive	no	false	Used to specify whether to support Hive	
-d / --directly	no	false	True for console insertion; false for sst import (TODO)	
-D / --dry	no	false	Check if the configuration file is correct	

PERFORMANCE

It takes about four minutes (i.e. 400k QPS) to input 100 million rows (each row contains three fields, each batch contains 64 rows) into three nodes (56 core, 250G memory, 10G network, SSD).

Last update: April 15, 2021

3.5.2 Data Export

Dump Tool

Dump Tool is a single-machine off-line data dumping tool that can be used to dump or count data with specified conditions.

HOW TO GET

The source code of the dump tool is under `nebula/src/tools/db_dump` directory. You can use command `make db_dump` to compile it. Before using this tool, you can use the SHOW HOSTS statement in the **Nebula Graph** console to check the distribution of the partitions. Also, you can use the `vertex_id % partition_num - 1` formula to calculate in which partition the vertex's corresponding key is located.

> **NOTE**: The *Dump Tool* is located in the rpm package and its directory is `nebula/bin/`. Because this tool dumps data by directly opening the RocksDB, you need to use it offline. Stop the storaged process and make sure that the metad service is started. Please refer to the following section for detailed usage.

HOW TO USE

The `db_dump` command displays information about how to use the dump tool. Parameter `space` is required. Parameters `db_path` and `meta_server` both have default values and you can configure them based on your actual situation. You can combine parameters `vids`, `parts`, `tags` and `edges` arbitrarily to dump the data you want.

```
  ./db_dump --space=<space name>

required:
       --space=<space name>
        # A space name must be given.

optional:
       --db_path=<path to rocksdb>
         # Path to the RocksDB data directory. If nebula was installed in `/usr/local/nebula`,
         # the db_path would be /usr/local/nebula/data/storage/nebula/
         # Default: ./

       --meta_server=<ip:port,...>
         # A list of meta severs' ip:port separated by comma.
         # Default: 127.0.0.1:45500

       --mode= scan | stat
         # scan: print to screen when records meet the condition, and also print statistics to screen in final.
         # stat: print statistics to screen.
         # Default: scan

       --vids=<list of vid>
         # A list of vids separated by comma. This parameter means vertex_id/edge_src_id
         # Would scan the whole space's records if it is not given.

       --parts=<list of partition id>
         # A list of partition ids separated by comma.
         # Would output all partitions if it is not given.

       --tags=<list of tag name>
         # A list of tag name separated by comma.

       --edges=<list of edge name>
         # A list of edge name separated by comma.

       --limit=<N>
         # A positive number that limits the output.
         # Would output all if set to 0 or negative.
         # Default: 1000
```

Following is an example:

```
// Specify a space to dump data
./db_dump --space=space_name

// Specify space, db_path, meta_server
./db_dump --space=space_name --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513

// Set mode to stat, only stats are returned, no data is printed
./db_dump --space=space_name --mode=stat --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513

// Specify vid to dump the vertex and the edges sourcing from it
./db_dump --space=space_name --mode=stat --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513 --vids=123,456

// Specify tag and dump vertices with the tag
./db_dump --space=space_name --mode=stat --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513 --tags=tag1,tag2
```

The returned data format:

```
// vertices, key: part_id, vertex_id, tag_name, value: <prop_list>
[vertex] key: 1, 0, poi value:mid:
0,8191765721868409651,8025713627522363385,1993089399535188613,3926276052777355165,5123607763506443893,2990089379644866415,poi_name_0,□□,□□,
30.2824,120.016,poi_stat_0,poi_fc_0,poi_sc_0,0,poi_star_0,

// edges, key: part_id, src_id, edge_name, ranking, dst_id, value: <prop_list>
[edge] key: 1, 0, consume_poi_reverse, 0, 656384 value:mid:656384,mid:0,7.19312,mid:656384,3897457441682646732,mun:656384,4038264117233984707,dun:656384,empe:
656384,mobile:656384,gender:656384,age:656384,rs:656384,fpd:656384,0.75313,1.34433,fpd:656384,0.03567,7.56212,

// stats
=======================================================
COUNT: 10           # total number of data dumped
VERTEX COUNT: 1     # number of vertices dumped
EDGE COUNT: 9       # number of edges dumped
TAG STATISTICS:     # number of tags dumped
        poi : 1
EDGE STATISTICS:    # number of edge types dumped
        consume_poi_reverse : 9
=======================================================
```

Last update: April 8, 2021

3.5.3 Storage Balance Usage

Nebula Graph's services are composed of three parts: graphd, storaged and metad. The **balance** in this document focuses on the operation of storage.

Currently, storage can be scaled horizontally by the command `balance`. There are two kinds of balance command, one is to move data, which is `BALANCE DATA`; the other one only changes the distribution of leader partition to balance load without moving data, which is `BALANCE LEADER`.

Balance data

Let's use an example to expand the cluster from 3 instances to 8 to show how to BALANCE DATA.

STEP 1 PREREQUISITES

Deploy a cluster with three replicas, one graphd, one metad and three storaged. Check cluster status using command `SHOW HOSTS`.

Step 1.1

```
nebula> SHOW HOSTS

=================================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=================================================================================================
| 192.168.8.210 | 34600 | online | 0            | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 0            | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 0            | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
```

Explanations on the returned results:

- **IP** and **Port** are the present storage instance, the cluster starts with three storaged instances (192.168.8.210:34600, 192.168.8.210:34700, 192.168.8.210:34500) without any data.
- **Status** shows the state of the present instance, currently there are two kind of states, i.e. online/offline. When a machine is out of service, metad will turn it to offline if no heart beat received for certain time threshold. The default threshold is 10 minutes and can be changed in parameter `expired_threshold_sec` when starting metad service.
- **Leader count** shows RAFT leader number of the present process.
- **Leader distribution** shows how the present leader is distributed in each graph space. No space is created for now.
- **Partition distribution** shows how many partitions are served by each host.

Step 1.2

Create a graph space named **test** with 100 partition and 3 replicas.

```
nebula> CREATE SPACE test(PARTITION_NUM=100, REPLICA_FACTOR=3)
```

Get the new partition distribution by the command `SHOW HOSTS`.

```
nebula> SHOW HOSTS
=================================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=================================================================================================
| 192.168.8.210 | 34600 | online | 0            | test: 0             | test: 100              |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 52           | test: 52            | test: 100              |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 48           | test: 48            | test: 100              |
-------------------------------------------------------------------------------------------------
```

STEP 2 ADD NEW HOSTS INTO THE CLUSTER

Now, add some new hosts (storaged instances) into the cluster.

Again, show the new status using command `SHOW HOSTS`. You can see there are already eight hosts in serving. But no partition is running on the new hosts.

```
nebula> SHOW HOSTS
=============================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=============================================================================================
| 192.168.8.210 | 34600 | online | 0            | test: 0             | test: 100              |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34900 | online | 0            | No valid partition  | No valid partition     |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 35940 | online | 0            | No valid partition  | No valid partition     |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34920 | online | 0            | No valid partition  | No valid partition     |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 44920 | online | 0            | No valid partition  | No valid partition     |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 52           | test: 52            | test: 100              |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 48           | test: 48            | test: 100              |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34800 | online | 0            | No valid partition  | No valid partition     |
---------------------------------------------------------------------------------------------
```

STEP 3 DATA MIGRATION

Check the current balance plan ID using command `BALANCE DATA` if the cluster is balanced. Otherwise, a new plan ID will be generated by the command.

```
nebula> BALANCE DATA
==============
| ID         |
==============
| 1570761786 |
--------------
```

Check the progress of balance using command `BALANCE DATA $id`.

```
nebula> BALANCE DATA 1570761786
================================================================================
| balanceId, spaceId:partId, src->dst                          | status      |
================================================================================
| [1570761786, 1:1, 192.168.8.210:34600->192.168.8.210:44920]  | succeeded   |
--------------------------------------------------------------------------------
| [1570761786, 1:1, 192.168.8.210:34700->192.168.8.210:34920]  | succeeded   |
--------------------------------------------------------------------------------
| [1570761786, 1:1, 192.168.8.210:34500->192.168.8.210:34800]  | succeeded   |
--------------------------------------------------------------------------------
| [1570761786, 1:2, 192.168.8.210:34600->192.168.8.210:44920]  | in progress |
--------------------------------------------------------------------------------
| [1570761786, 1:2, 192.168.8.210:34700->192.168.8.210:34920]  | in progress |
--------------------------------------------------------------------------------
| [1570761786, 1:2, 192.168.8.210:34500->192.168.8.210:34800]  | in progress |
--------------------------------------------------------------------------------
| [1570761786, 1:3, 192.168.8.210:34600->192.168.8.210:44920]  | succeeded   |
--------------------------------------------------------------------------------
...
| Total:189, Succeeded:170, Failed:0, In Progress:19, Invalid:0 | 89.947090%  |
```

Explanations on the returned results:

- The first column is a specific balance task. Take 1570761786, 1:88, 192.168.8.210:34700->192.168.8.210:35940 for example
  - **1570761786** is the balance ID
  - **1:88**, 1 is the spaceId, 88 is the moved partId
  - **192.168.8.210:34700->192.168.8.210:3594**, moving data from 192.168.8.210:34700 to 192.168.8.210:35940
- The second column shows the state (result) of the task, there are four states:
  - Succeeded
  - Failed
  - In progress
  - Invalid

The last row is the summary of the tasks. Some partitions are yet to be migrated.

STEP 4 MIGRATION CONFIRMATION

In most cases, data migration will take hours or even days. During the migration, **Nebula Graph** services are not affected. Once migration is done, the progress will show 100%. You can retry `BALANCE DATA` to fix a failed task. If it can't be fixed after several attempts, please contact us at GitHub.

Now, you can check partition distribution using command `SHOW HOSTS` when balance completed.

```
nebula> SHOW HOSTS
=============================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=============================================================================================
| 192.168.8.210 | 34600 | online | 3            | test: 3             | test: 37               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34900 | online | 0            | test: 0             | test: 38               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 35940 | online | 0            | test: 0             | test: 37               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34920 | online | 0            | test: 0             | test: 38               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 44920 | online | 0            | test: 0             | test: 38               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 35           | test: 35            | test: 37               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 24           | test: 24            | test: 37               |
---------------------------------------------------------------------------------------------
| 192.168.8.210 | 34800 | online | 38           | test: 38            | test: 38               |
---------------------------------------------------------------------------------------------
```

Now partitions and data are evenly distributed on the machines.

Balance stop

`BALANCE DATA STOP` command stops the running balance data plan. If there is no running balance plan, an error is thrown. If there is a running plan, the related plan ID is returned.

> Since each balance plan includes several balance tasks, `BALANCE DATA STOP` doesn't stop the started tasks , but rather cancel the subsequent tasks. The started tasks will continue until the executions are completed.

Input `BALANCE DATA $id` after `BALANCE DATA STOP` to check the status of the stopped balance plan.

After all the tasks being executed are completed, rerun the `BALANCE DATA` command to restart balance.

If there are failed tasks in the stopped plan, the plan will continue. Otherwise, if all the tasks are succeed, a new balance plan is created and executed.

Batch Scale in

Nebula supports specifying hosts that need to go offline to conduct batch scale in. The syntax is `BALANCE DATA REMOVE $host_list`. For example, statement `BALANCE DATA REMOVE 192.168.0.1:50000,192.168.0.2:50000` removes two hosts, i.e. 192.168.0.1:50000 192.168.0.2:50000, during the balance process.

> If replica number cannot meet the requirement after removing (for example, the number of remaining hosts is less than the number of replicas or when one of the three replica is offline, one of the remaining two replicas is required to be removed), **Nebula Graph** will reject the balance request and return an error code.

Balance leader

Command `BALANCE DATA` only migrates partitions. But the leader distribution remains unbalanced, which means old hosts are overloaded, while the new ones are not fully used. Redistribute RAFT leader using the command `BALANCE LEADER`.

```
nebula> BALANCE LEADER
```

Seconds later, check the results using the command `SHOW HOSTS`. The RAFT leaders are distributed evenly over all the hosts in the cluster.

```
nebula> SHOW HOSTS
=============================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
```

```
==========================================================================================
| 192.168.8.210 | 34600 | online | 13           | test: 13            | test: 37              |
------------------------------------------------------------------------------------------
| 192.168.8.210 | 34900 | online | 12           | test: 12            | test: 38              |
------------------------------------------------------------------------------------------
| 192.168.8.210 | 35940 | online | 12           | test: 12            | test: 37              |
------------------------------------------------------------------------------------------
| 192.168.8.210 | 34920 | online | 12           | test: 12            | test: 38              |
------------------------------------------------------------------------------------------
| 192.168.8.210 | 44920 | online | 13           | test: 13            | test: 38              |
------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 12           | test: 12            | test: 37              |
------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 13           | test: 13            | test: 37              |
------------------------------------------------------------------------------------------
| 192.168.8.210 | 34800 | online | 13           | test: 13            | test: 38              |
------------------------------------------------------------------------------------------
```

Last update: April 8, 2021

3.5.4 Cluster Snapshot

Create Snapshot

The `CREATE SNAPSHOT` command creates a snapshot at the current point in time for the whole cluster. The snapshot name is composed of the timestamp of the meta server. If snapshot creation fails in the current version, you must use the `DROP SNAPSHOT` to clear the invalid snapshots.

The current version does not support creating snapshot for the specified graph spaces, and executing `CREATE SNAPSHOT` creates a snapshot for all graph spaces in the cluster. For example:

```
nebula> CREATE SNAPSHOT;
Execution succeeded (Time spent: 22892/23923 us)
```

Show Snapshots

The command `SHOW SNAPSHOT` looks at the states (VALID or INVALID), names and the IP addresses of all storage servers when the snapshots are created in the cluster. For example:

```
nebula> SHOW SNAPSHOTS;
============================================================
| Name                          | Status | Hosts           |
============================================================
| SNAPSHOT_2019_12_04_10_54_36 | VALID  | 127.0.0.1:77833 |
------------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_42 | VALID  | 127.0.0.1:77833 |
------------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_44 | VALID  | 127.0.0.1:77833 |
------------------------------------------------------------
```

Delete Snapshot

The `DROP SNAPSHOT` command deletes a snapshot with the specified name, the syntax is:

```
DROP SNAPSHOT <snapshot-name>
```

You can get the snapshot names with the command `SHOW SNAPSHOTS`. `DROP SNAPSHOT` can delete both valid snapshots and invalid snapshots that failed during creation. For example:

```
nebula> DROP SNAPSHOT SNAPSHOT_2019_12_04_10_54_36;
nebula> SHOW SNAPSHOTS;
============================================================
| Name                          | Status | Hosts           |
============================================================
| SNAPSHOT_2019_12_04_10_54_42 | VALID  | 127.0.0.1:77833 |
------------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_44 | VALID  | 127.0.0.1:77833 |
------------------------------------------------------------
```

Now the deletes snapshot is not in the show snapshots list.

Tips

- When the system structure changes, it is better to create a snapshot immediately. For example, when you add host, drop host, create space, drop space or balance.
- The current version does not support automatic garbage collection for the failed snapshots in creation. We will develop cluster checker in meta server to check the cluster state via asynchronous threads and automatically collect the garbage files in failure snapshot creation.
- The current version does not support customized snapshot directory. The snapshots are created in the `data_path/nebula` directory by default.
- The current version does not support snapshot restore. Users need to write a shell script based on their actual productions to restore snapshots. The implementation logic is rather simple, you copy the snapshots of the engine servers to the specified folder, set this folder to `data_path/`, then start the cluster.

Last update: April 8, 2021

3.5.5 Job Manager

The job here refers to the long tasks running at the storage layer. For example, `compact` and `flush`. The manager means to manage the jobs. For example, you can run, show, stop and recover jobs.

Statements List

SUBMIT JOB COMPACT

The `SUBMIT JOB COMPACT` command triggers the long-term `RocksDB compact` operation. The example returns the results as follows:

```
nebula> SUBMIT JOB COMPACT;
==============
| New Job Id |
==============
| 40         |
--------------
```

See here to modify the default compact thread number.

SUBMIT JOB FLUSH

The `SUBMIT JOB FLUSH` command writes the RocksDB memfile in memory to the hard disk.

```
nebula> SUBMIT JOB FLUSH;
==============
| New Job Id |
==============
| 2          |
--------------
```

SHOW JOB

List Single Job Information

The `SHOW JOB <job_id>` statement shows a job with certain ID and all its tasks. After a job arrives to Meta, Meta will split the job to tasks, and send them to storage.

```
nebula> SHOW JOB 40
=================================================================================================
| Job Id(TaskId) | Command(Dest)              | Status   | Start Time        | Stop Time         |
=================================================================================================
| 40             | flush basketballplayer     | finished | 12/17/19 17:21:30 | 12/17/19 17:21:30 |
-------------------------------------------------------------------------------------------------
| 40-0           | 192.168.8.5                | finished | 12/17/19 17:21:30 | 12/17/19 17:21:30 |
-------------------------------------------------------------------------------------------------
```

The above statement returns one to multiple rows, which is determined by the `storaged` number where the space is located.

What's in the returned results:

- `40` is the job ID
- `flush basketballplayer` indicates that a flush operation is performed on space basketballplayer
- `finished` is the job status, which indicates that the job execution is finished and successful. Other job status are Queue, running, failed and stopped
- `12/17/19 17:21:30` is the start time, which is initially empty(Queue). The value is set if and only if the job status is running.
- `12/17/19 17:21:30` is the stop time, which is empty when the job status is Queue or running. The value is set when the job status is finished, failed and stopped
- `40-0` indicated that the job ID is 40, the task ID is 0
- `192.168.8.5` shows which machine the job is running on
- `finished` is the job status, which indicates that the job execution is finished and successful. Other job status are Queue, running, failed and stopped
- `12/17/19 17:21:30` is the start time, which can never be empty because the initial status is running
- `12/17/19 17:21:30` is the end time, which is empty when the job status is running. The value is set when the job status is finished, failed and stopped

NOTE: There are five job states, i.e. QUEUE, RUNNING, FINISHED, FAILED, STOPPED. Status switching is described below:

```
Queue -- running -- finished -- removed
     \          \                /
      \          \ -- failed -- /
       \          \            /
        \ ---------- stopped -/
```

List All Jobs

The `SHOW JOBS` statement lists all the jobs that are not expired. The default job expiration time is one week. You can change it with meta flag `job_expired_secs`.

```
nebula> SHOW JOBS
=================================================================================
| Job Id | Command       | Status   | Start Time         | Stop Time         |
=================================================================================
| 22     | flush test2   | failed   | 12/06/19 14:46:22 | 12/06/19 14:46:22 |
---------------------------------------------------------------------------------
| 23     | compact test2 | stopped  | 12/06/19 15:07:09 | 12/06/19 15:07:33 |
---------------------------------------------------------------------------------
| 24     | compact test2 | stopped  | 12/06/19 15:07:11 | 12/06/19 15:07:20 |
---------------------------------------------------------------------------------
| 25     | compact test2 | stopped  | 12/06/19 15:07:13 | 12/06/19 15:07:24 |
---------------------------------------------------------------------------------
```

For details on the returned results, please refer to the previous section List Single Job Information.

STOP JOB

The `STOP JOB` statement stops jobs that are not finished.

```
nebula> STOP JOB 22
========================
| STOP Result          |
========================
| stop 1 jobs 2 tasks |
------------------------
```

RECOVER JOB

The `RECOVER JOB` statement re-executes the failed jobs and returns the number of the recovered jobs.

```
nebula> RECOVER JOB
====================
| Recovered job num |
====================
| 5 job recovered   |
--------------------
```

FAQ

`SUBMIT JOB` uses HTTP port. Please check if the HTTP ports among the storages are normal. You can use the following command to debug.

```
curl "http://{storaged-ip}:12000/admin?space={test}&op=compact"
```

Last update: April 16, 2021

3.5.6 Compact

This document will walk you through the concept of Compact.

- The default RocksDB compact style. You can use the `UPDATE CONFIG` statement as follows to start or stop it. It merges sst files in small scale during data writing to speed up the data reading in a short time. When enabled, this compaction is automatically performed during the daytime.

```
nebula> UPDATE CONFIGS storage:rocksdb_column_family_options = {disable_auto_compactions = true};
```

By default, the `disable_auto_compactions` parameter is set to `false`. Before data import, we recommend that you set the parameter to `true`. When the data is imported, you must set the parameter to its default value, and then run the `SUBMIT JOB COMPACT` command to perform the customized compaction.

- The customized compact style for **Nebula Graph**. You can run the `SUBMIT JOB COMPACT` command to start it. You can use it to perform large scale background operations such as sst files merging in large scale or TTL. This kind of compact is usually performed after midnight.

In addition, you can modify the number of threads in both methods by the following command.

```
nebula> UPDATE CONFIGS storage:rocksdb_db_options  = \
        { max_subcompactions = 4, max_background_jobs = 4};
```

Last update: April 8, 2021

3.6 Monitoring and Statistics

3.6.1 Metrics Exposer

See here.

Last update: April 8, 2021

3.6.2 Meta Metrics

Introduction

Currently, **Nebula Graph** supports obtaining the basic performance metrics for the meta service via HTTP.

Each performance metric consists of three parts, namely `<counter_name>.<statistic_type>.<time_range>`.

COUNTER NAMES

Each counter name is composed of the interface name and the counter name. Meta service only counts the heartbeat. Currently, the supported interfaces are:

```
meta_heartbeat_qps
meta_heartbeat_error_qps
meta_heartbeat_latency
```

STATISTICS TYPE

Currently supported types are SUM, COUNT, AVG, RATE and P quantiles (P99, P999, ..., P999999). Among which:

- Metrics have suffixes `_qps` and `_error_qps` support SUM, COUNT, AVG, RATE but don't support P quantiles.
- Metrics have suffixes `_latency` support SUM, COUNT, AVG, RATE, and P quantiles.

TIME RANGE

Currently, the supported time ranges are 60s, 600s, and 3600s, which correspond to the last minute, the last ten minutes, and the last hour till now.

Obtain the Corresponding Metrics via HTTP Interface

Here are some examples:

```
meta_heartbeat_qps.avg.60         // the average QPS of the heart beat in the last minute
meta_heartbeat_error_qps.count.60   // the total errors occurred of the heart beat in the last minute
meta_heartbeat_latency.avg.60     // the average latency of the heart beat in the last minute
```

Assume that a **Nebula Graph** meta service is started locally, and the `ws_http_port` port number is set to 11000 when starting. It is sent through the **GET** interface of HTTP. The method name is **get_stats**, and the parameter is stats plus the corresponding metrics name. Here's an example of getting metrics via the HTTP interface:

```
# obtain a metrics
curl -G "http://127.0.0.1:11000/get_stats?stats=meta_heartbeat_qps.avg.60"
# meta_heartbeat_qps.avg.60=580

# obtain multiple metrics at the same time
curl -G "http://127.0.0.1:11000/get_stats?stats=meta_heartbeat_qps.avg.60,meta_heartbeat_error_qps.avg.60"
# meta_heartbeat_qps.avg.60=537
# meta_heartbeat_error_qps.avg.60=579

# obtain multiple metrics at the same time and return in json format
curl -G "http://127.0.0.1:11000/get_stats?stats=meta_heartbeat_qps.avg.60,meta_heartbeat_error_qps.avg.60&returnjson"
# [{"value":533,"name":"meta_heartbeat_qps.avg.60"},{"value":574,"name":"meta_heartbeat_error_qps.avg.60"}]

# obtain all the metrics
curl -G "http://127.0.0.1:11000/get_stats?stats"
# or
curl -G "http://127.0.0.1:11000/get_stats"
```

Last update: April 8, 2021

3.6.3 Storage Metrics

Introduction

Currently, **Nebula Graph** supports obtaining the basic performance metrics for the storage service via HTTP.

Each performance metric consists of three parts, namely `<counter_name>.<statistic_type>.<time_range>`, details are introduced as the follows.

COUNTER NAMES

Each counter name is composed of the interface name and the counter name. Currently, the supported interfaces are:

```
storage_vertex_props // obtain properties of a vertex
storage_edge_props // obtain properties of an edge
storage_add_vertex // insert a vertex
storage_add_edge // insert an edge
storage_del_vertex // delete a vertex
storage_update_vertex // update properties of a vertex
storage_update_edge // update properties of an edge
storage_get_kv // read kv pair
storage_put_kv // put kv pair
storage_get_bound // internal use only
```

Each interface has three metrics, namely latency (in the units of us), QPS and QPS with errors. The suffixes are as follows:

```
_latency
_qps
_error_qps
```

The complete metric concatenates the interface name with the corresponding metric, such as `storage_add_vertex_latency`, `storage_add_vertex_qps` and `storage_add_vertex_error_qps`, representing the latency, QPS, and the QPS with errors of inserting a vertex, respectively.

STATISTICS TYPE

Currently supported types are SUM, COUNT, AVG, RATE, and P quantiles (P99, P999, ..., P999999). Among which:

- Metrics have suffixes `_qps` and `_error_qps` support SUM, COUNT, AVG, RATE but don't support P quantiles.
- Metrics have suffixes `_latency` support SUM, COUNT, AVG, RATE, and P quantiles.

TIME RANGE

Currently, the supported time ranges are 60s, 600s, and 3600s, which correspond to the last minute, the last ten minutes, and the last hour till now.

Obtain the Corresponding Metrics via HTTP Interface

According to the above introduction, you can make a complete metrics name, here are some examples:

```
storage_add_vertex_latency.avg.60   // the average latency of inserting a vertex in the last minute
storage_get_bound_qps.rate.600  // obtain neighbor's QPS in the last ten minutes
storage_update_edge_error_qps.count.3600  // errors occurred in updating an edge in the last hour
```

Assume that a **Nebula Graph** storage service is started locally, and the `ws_http_port` port number is set to 12000 when starting. It is sent through the **GET** interface of HTTP. The method name is **get_stats**, and the parameter is stats plus the corresponding metrics name. Here's an example of getting metrics via the HTTP interface:

```
# obtain a metrics
curl -G "http://127.0.0.1:12000/get_stats?stats=storage_vertex_props_qps.rate.60"
# storage_vertex_props_qps.rate.60=2674

# obtain multiple metrics at the same time
curl -G "http://127.0.0.1:12000/get_stats?stats=storage_vertex_props_qps.rate.60,storage_vertex_props_latency.avg.60"
# storage_vertex_props_qps.rate.60=2638
# storage_vertex_props_latency.avg.60=812

# obtain multiple metrics at the same time and return in json format
curl -G "http://127.0.0.1:12000/get_stats?stats=storage_vertex_props_qps.rate.60,storage_vertex_props_latency.avg.60&returnjson"
# [{"value":2723,"name":"storage_vertex_props_qps.rate.60"},{"value":804,"name":"storage_vertex_props_latency.avg.60"}]
```

```
# obtain all the metrics
curl -G "http://127.0.0.1:12000/get_stats?stats"
# or
curl -G "http://127.0.0.1:12000/get_stats"
```

Last update: April 8, 2021

3.6.4 Graph Metrics

Introduction

Currently, **Nebula Graph** supports obtaining the basic performance metric for the graph service via HTTP.

Each performance metrics consists of three parts, namely `<counter_name>.<statistic_type>.<time_range>`.

COUNTER NAMES

Each counter name is composed of the interface name and the counter name. Currently, the supported interfaces are:

```
graph_storageClient // Requests sent via storageClient, when sending requests to multiple storages concurrently, counted as one
graph_metaClient // Requests sent via metaClient
graph_graph_all // Requests sent by the client to the graph, when a request contains multiple queries, counted as one
graph_insertVertex // Insert a vertex
graph_insertEdge // Insert an edge
graph_deleteVertex // Delete a vertex
graph_deleteEdge // Delete an edge // Not supported yet
graph_updateVertex // Update properties of a vertex
graph_updateEdge // Update properties of an edge
graph_go // Execute the go command
graph_findPath // Find the shortest path or the full path
graph_fetchVertex // Fetch the vertex's properties. Only count the commands executed rather than the total number of fetched vertices.
graph_fetchEdge // Fetch the edge's properties. Only count the commands executed rather than the total number of fetched edges.
```

Each interface has three metrics, namely latency (in the units of us), QPS and QPS with errors. The suffixes are as follows:

```
_latency
_qps
_error_qps
```

The complete metric concatenates the interface name with the corresponding metric, such as `graph_insertVertex_latency`, `graph_insertVertex_qps` and `graph_insertVertex_error_qps`, representing the latency of inserting a vertex, QPS and the QPS with errors, respectively.

STATISTICS TYPE

Currently supported types are SUM, COUNT, AVG, RATE and P quantiles (P99, P999, ..., P999999). Among which:

- Metrics have suffixes `_qps` and `_error_qps` support SUM, COUNT, AVG, RATE but don't support P quantiles.
- Metrics have suffixes `_latency` support SUM, COUNT, AVG, RATE, and P quantiles.

TIME RANGE

Currently, the supported time ranges are 60s, 600s, and 3600s, which correspond to the last minute, the last ten minutes, and the last hour till now.

Obtain the Corresponding Metrics via HTTP Interface

According to the above introduction, you can make a complete metrics name. Here are some examples:

```
graph_insertVertex_latency.avg.60   // the average latency of successfully inserting a vertex in the last minute
graph_updateEdge_error_qps.count.3600  // total number of failures in updating an edge in the last hour
```

Assume that a graph service is started locally, and the `ws_http_port` port number is set to 13000 when starting. It is sent through the **GET** interface of HTTP. The method name is **get_stats**, and the parameter is stats plus the corresponding metrics name. Here's an example of getting metrics via the HTTP interface:

```
# obtain a metrics
curl -G "http://127.0.0.1:13000/get_stats?stats=graph_insertVertex_qps.rate.60"
# graph_insertVertex_qps.rate.60=3069

# obtain multiple metrics at the same time
curl -G "http://127.0.0.1:13000/get_stats?stats=graph_insertVertex_qps.rate.60, graph_deleteVertex_latency.avg.60"
# graph_insertVertex_qps.rate.60=3069
# graph_deleteVertex_latency.avg.60=837

# obtain multiple metrics at the same time and return in json format
curl -G "http://127.0.0.1:13000/get_stats?stats=graph_insertVertex_qps.rate.60, graph_deleteVertex_latency.avg.60&returnjson"
# [{"value":2373,"name":"graph_insertVertex_qps.rate.60"},{"value":760,"name":"graph_deleteVertex_latency.avg.60"}]
```

```
# obtain all the metrics
curl -G "http://127.0.0.1:13000/get_stats?stats"
# or
curl -G "http://127.0.0.1:13000/get_stats"
```

Last update: April 8, 2021

3.6.5 RocksDB statistics

Nebula Graph uses RocksDB as its underlying storage. The purpose of this document is to teach you how to collect and display the RocksDB statistics for **Nebula Graph**.

Enabling the RocksDB statistics

The RocksDB Statistics function is disabled by default. If you want to enable the RocksDB statistics function, you need to:

- modify the `--enable_rocksdb_statistics` parameter to `true` in the storage configuration file `storaged.conf`. The default configuration file directory is `/home/user/nebula/build/install/etc`.
- restart the services to make the modifications take effect.

When the function is enabled, the statistics is dumped to the log file of each DB service regularly.

Getting the RocksDB statistics

You can use the built-in web interface in the storage service to get the statistics. There are three methods to get the RocksDB statistics by using the web service:

1. Get all the statistics.
2. Get information for a specified entry.
3. Support returning the Json format result.

Examples

Get all the RocksDB statistics by using the following command:

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats"
```

For example:

```
curl -L "http://172.28.2.1:12000/rocksdb_stats"

rocksdb.blobdb.blob.file.bytes.read=0
rocksdb.blobdb.blob.file.bytes.written=0
rocksdb.blobdb.blob.file.bytes.synced=0
...
```

Get part of the RocksDB statistics by using the following command:

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats?stats=${stats_name}"
```

For example:

```
curl -L "http://172.28.2.1:12000/rocksdb_stats?stats=rocksdb.bytes.read,rocksdb.block.cache.add"

rocksdb.block.cache.add=14
rocksdb.bytes.read=1632
```

Get part of the RocksDB statistics in JSON format by using the following command:

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats?stats=${stats_name}.&returnjson"
```

For example:

```
curl -L "http://172.28.2.1:12000/rocksdb_stats?stats=rocksdb.bytes.read,rocksdb.block.cache.add&returnjson"

[{"value":14,"name":rocksdb.block.cache.add},{"value":1632,"name":"rocksdb.bytes.read"}]
```

Last update: April 8, 2021

3.7 Development and API

3.7.1 Supported Clients by Nebula Graph

Currently, **Nebula Graph** supports the following clients:

- Go Client
- Python Client
- Java Client

Storage Service Clients

You can also access the storage service and the key-value service directly:

- key-value api

Besides, find more examples here.

Last update: April 8, 2021

4. Data Migration

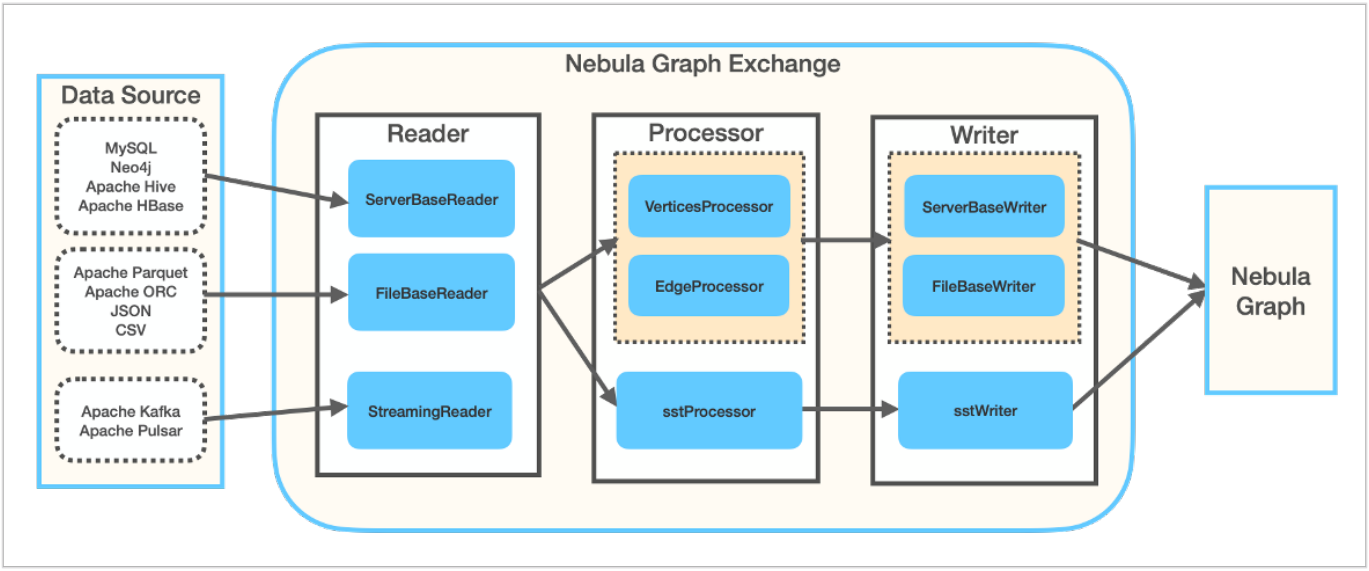
4.1 Nebula Exchange

4.1.1 About Nebula Exchange

What is Nebula Exchange

Nebula Exchange v1.x (Exchange v1.x or Exchange in short) is an Apache Spark™ application. It can be used to migrate data from a cluster in a distributed environment to a Nebula Graph v1.x cluster. It supports processing different formats of batch data and streaming data.

Exchange is composed of Reader, Processor, and Writer. Reader reads data of different sources and creates DataFrame. Processor traverses every row of the DataFrame and obtains the values for each column according to the mapping of the `fields` in the configuration file. And then after the specified rows of data to be batch processed are traversed, Writer writes the obtained data into Nebula Graph concurrently. This figure shows how the data is transformed and transferred in Exchange v1.x.



SCENARIOS

You can use Exchange in these scenarios:

- Converting streaming data from Kafka or Pulsar platforms to vertex or edge data of property graphs and importing them into Nebula Graph. For example, log files, online shopping data, in-game player activities, social networking information, financial trading services, geospatial services, or telemetry data from connected devices or instruments in the data center.
- Converting batch data (such as data in a certain period of time) from a relational database (such as MySQL) or a distributed file system (such as HDFS) into vertex or edge data of property graphs, and importing them into Nebula Graph.
- Converting a large amount of data into SST files and then importing them into Nebula Graph.

FEATURES

Exchange has these features:

- Adaptable: Exchange supports importing data from different sources into Nebula Graph, which is convenient for you to migrate data.
- SST files supported: Exchange supports converting data from different sources into SST files for data import.

  NOTE: Importing SST files with Exchange v1.x is supported in Linux only.
- Resuming broken transfer: Exchange supports resuming an interrupted transfer from a broken point during the data import process, which saves your time and improves efficiency.

  NOTE: Exchange v1.x supports resuming broken transfer for Neo4j only.
- Asynchronous: Exchange enables you to set an insertion statement for the source and sends it to the Graph Service of Nebula Graph for data insertion.
- Flexible: Exchange supports importing multiple types of vertices and edges of different sources or formats simultaneously.
- Statistics: Exchange uses the accumulator in Apache Spark™ to count the successes and failures during the insertion process.
- Easy to use and user-friendly: Exchange supports HOCON (Human-Optimized Config Object Notation) configuration file format, which is object-oriented, and easy to understand and operate.

SUPPORTED DATA SOURCES

You can use Exchange v1.x to convert data of these sources into vertex and edge data and then import them to Nebula Graph v1.x:

- Data of different formats stored on HDFS, including:
  - Apache Parquet
  - Apache ORC
  - JSON
  - CSV
- Apache HBase™
- Data warehouses: HIVE
- Graph databases: Neo4j 2.4.5-M1. Resuming transfer from a broken point is supported for Neo4j data only.
- Relational databases: MySQL
- Stream processing platforms: Apache Kafka®
- Messaging and streaming platforms: Apache Pulsar 2.4.5

Last update: April 8, 2021

Limitations

This article introduces the limitations of Exchange v1.x.

SUPPORTED NEBULA GRAPH VERSIONS

Exchange v1.x supports Nebula Graph v1.x only. If you are using Nebula Graph v2.x, please use Nebula Exchange v2.x.

SUPPORTED OPERATION SYSTEMS

You can use Exchange v1.x in these operation systems:

- CentOS 7
- macOS

NOTE: Importing SST files with Exchange v1.x is supported in Linux only.

SOFTWARE DEPENDENCIES

To make sure that Exchange v1.x works properly, make sure that these software applications are installed in your machine:

- Apache Spark: 2.3.0 or later versions
- Java: 1.8
- Scala: 2.10.7, 2.11.12, or 2.12.10

In these scenarios, Hadoop Distributed File System (HDFS) must be deployed:

- Importing data from HDFS to Nebula Graph
- Importing SST files into Nebula Graph

Last update: April 8, 2021

Glossary

This article gives explanations of some necessary terminologies in this user guide.

- Nebula Exchange: Referred to as Exchange v1.x or Exchange in this user guide. It is a Spark application based on Apache Spark™ for batch or stream processing data migration. It supports converting data from different sources into vertex and edge data that can be recognized by Nebula Graph, and then concurrently importing data into Nebula Graph.
- Apache Spark™: A fast and general computing engine designed for large-scale data processing. It is an open-source project of Apache Software Foundation.
- Driver Program: Referred to as driver in this user guide. It is a program that runs the `main` function of an application and creates a new `SparkContext` instance.

Last update: April 8, 2021

FAQ

What version of Nebula Graph does Exchange v1.x support?

Read Limitations to get the latest information about supported Nebula Graph versions.

What is the difference between Exchange and Spark Writer?

Both are Spark applications, and Exchange is based on Spark Writer. Both of them are designed for the migration of data into a Nebula Graph cluster in a distributed environment, but the later maintenance work will focus on Exchange. Compared with Spark Writer, Exchange has the following improvements:

- Supporting more data sources, such as MySQL, Neo4j, HIVE, HBase, Kafka, and Pulsar.
- Some problems with Spark Writer were fixed. For example, by default Spark reads source data from HDFS as strings, which is probably different from your graph schema defined in Nebula Graph. Exchange supports automatically matching and converting data types. With it, when a non-string data type is defined in Nebula Graph, Exchange converts the strings into data of the required data type.

Last update: April 8, 2021

4.1.2 Compile Exchange

To compile Exchange, follow these steps:

1. Clone source code from the `nebula-java` repository.

```
git clone -b v1.0 https://github.com/vesoft-inc/nebula-java.git
```

2. Change to the `nebula-java` directory and package Nebula Java 1.x.

```
cd nebula-java
mvn clean install -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

3. Change to the `tools/exchange` directory and compile Exchange.

```
cd tools/exchange
mvn clean package -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

After compiling, you can see the structure of the Exchange directory as follows.

```
.
├── README.md
├── dependency-reduced-pom.xml
├── pom.xml
├── scripts
│   ├── README.md
│   ├── mock_data.py
│   ├── pulsar_producer.py
│   ├── requirements.txt
│   └── verify_nebula.py
├── src
│   └── main
│       ├── resources
│       ├── scala
│       └── test
└── target
    ├── classes
    │   ├── application.conf
    │   ├── com
    │   ├── server_application.conf
    │   └── stream_application.conf
    ├── classes.timestamp
    ├── exchange-1.x.y-javadoc.jar
    ├── exchange-1.x.y-sources.jar
    ├── exchange-1.x.y.jar
    ├── generated-test-sources
    │   └── test-annotations
    ├── maven-archiver
    │   └── pom.properties
    ├── maven-status
    │   └── maven-compiler-plugin
    ├── original-exchange-1.x.y.jar
    ├── site
    │   └── scaladocs
    ├── test-classes
    │   └── com
    └── test-classes.timestamp
```

In the `target` directory, you can see the `exchange-1.x.y.jar` file.

> **NOTE**: The version of the JAR file depends on the releases of Nebula Java Client. You can find the latest versions on the Releases page of the nebula-java repository.

To import data, you can refer to the example configuration in the `target/classes/application.conf`, `target/classes/server_application.conf`, and `target/classes/stream_application.conf` files.

Last update: April 8, 2021

4.1.3 Examples

Use Exchange

This article introduces the generally-used procedure on how to use Exchange to import data from a specified source to Nebula Graph.

PREREQUISITES

To import data with Exchange, do a check of these:

- Nebula Graph is deployed and started. Get the information:
  - IP addresses and ports of the Graph Service and the Meta Service.
  - A Nebula Graph account with the privilege of writing data and its password.
- Exchange is compiled. For more information, see Compile Exchange.
- Spark is installed.
- Get the necessary information for schema creation in Nebula Graph, including tags and edge types.

PROCEDURE

To import data from a source to Nebula Graph, follow these steps:

1. Create a graph space and a schema in Nebula Graph.
2. (Optional) Process the source data. For example, to import data from a Neo4j database, create indexes for the specified tags in Neo4j to export the data from Neo4j more quickly.
3. Edit the configuration file for Spark, Nebula Graph, vertices, and edges.

   NOTE: After compiling of Exchange, refer to the example configuration files in the `nebula-java/tools/exchange/target/classes` directory for the configuration for different sources.
4. (Optional) Run the import command with the `-D` parameter to verify the configuration. For more information, see Import command parameters.
5. Run the import command to import data into Nebula Graph.
6. Verify the imported data in Nebula Graph.
7. (Optional) Create and rebuild indexes in Nebula Graph.

For more information, see the examples:

- Import data from Neo4j
- Import data from HIVE
- Import data from CSV files
- Import data from JSON files
- Import SST files

Last update: April 8, 2021

Import data from CSV files

This article uses an example to show how to use Exchange to import data from CSV files stored on HDFS into Nebula Graph.

If you want to import data from local CSV files into Nebula Graph v1.x, see Nebula Importer.

DATASET

In this article, the Social Network: MOOC User Action Dataset provided by Stanford Network Analysis Platform (SNAP) and 97 unique course names obtained from the public network are used as the sample dataset. The dataset contains:

- Two vertex types (`user` and `course`), 7,144 vertices in total.
- One edge type (`action`), 411,749 edges in total.

You can download the example dataset from the nebula-web-docker repository.

ENVIRONMENT

The practice is done in macOS. Here is the environment information:

- Hardware specifications:
  - CPU: 1.7 GHz Quad-Core Intel Core i7
  - Memory: 16 GB
- Spark 2.3.0, deployed in the Standalone mode
- Hadoop 2.9.2, deployed in the Pseudo-Distributed mode
- Nebula Graph v1.2.1, deployed with Docker Compose. For more information, see Deploy Nebula Graph with Docker Compose.

PREREQUISITES

To import data from CSV files on HDFS with Exchange v1.x, do a check of these:

- Exchange v1.x is compiled. For more information, see Compile Exchange v1.x. Exchange 1.2.1 is used in this example.
- Spark is installed.
- Hadoop is installed and started.
- Nebula Graph is deployed and started. Get the information:
  - IP addresses and ports of the Graph Service and the Meta Service.
  - A Nebula Graph account with the privilege of writing data and its password.
- Get the necessary information for schema creation in Nebula Graph, including tags and edge types.

PROCEDURE

Step 1. Create a schema in Nebula Graph

Analyze the data in the CSV files and follow these steps to create a schema in Nebula Graph:

1. Confirm the essential elements of the schema.

Elements	Names	Properties
Tag	`user`	`userId int`
Tag	`course`	`courseId int, courseName string`
Edge Type	`action`	`actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double`

2. In Nebula Graph, create a graph space named **csv** and create a schema.

```
-- Create a graph space named csv
CREATE SPACE csv(partition_num=10, replica_factor=1);

-- Choose the csv graph space
USE csv;

-- Create the user tag
CREATE TAG user(userId int);

-- Create the course tag
CREATE TAG course(courseId int, courseName string);

-- Create the action edge type
CREATE EDGE action (actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double);
```

For more information, see Quick Start of Nebula Graph Database User Guide.

Step 2. Prepare CSV files

Do a check of these:

1. The CSV files are processed to meet the requirements of the schema. For more information, see Quick Start of Nebula Graph Studio. **>NOTE**: Exchange supports importing CSV files with or without headers.
2. The CSV files must be stored in HDFS and get the file storage path.

Step 3. Edit configuration file

After compiling of Exchange, copy the `target/classes/application.conf` file and edit the configuration for CSV files. In this example, a new configuration file is named `CSV_ application.conf`. In this file, the vertex and edge related configuration is introduced as comments and all the items that are not used in this example are commented out. For more information about the Spark and Nebula related parameters, see Spark related parameters and Nebula Graph related parameters.

```
{
  # Spark related configuration
  spark: {
    app: {
      name: Spark Writer
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores {
      max: 16
    }
  }

  # Nebula Graph related configuration
  nebula: {
    address:{
      # Specifies the IP addresses and ports of the Graph Service and the Meta Service of Nebula Graph.
      # If multiple servers are used, separate the addresses with commas.
      # Format: "ip1:port","ip2:port","ip3:port".
      graph:["127.0.0.1:3699"]
      meta:["127.0.0.1:45500"]
    }

    # Specifies an account that has the WriteData privilege in Nebula Graph and its password.
    user: user
    pswd: password
```

```
  # Specifies a graph space name
  space: csv
  connection {
    timeout: 3000
    retry: 3
  }
  execution {
    retry: 3
  }
  error: {
    max: 32
    output: /tmp/errors
  }
  rate: {
    limit: 1024
    timeout: 1000
  }
}

# Process vertices
tags: [
  # Sets for the course tag
  {
    # Specifies a tag name defined in Nebula Graph.
    name: course
    type: {
      # Specifies the data source. csv is used.
      source: csv

      # Specifies how to import vertex data into Nebula Graph: client or sst.
      # For more information about importing sst files, see Import SST files (doc_to_do).
      sink: client
    }

    # Specifies the HDFS path of the CSV file.
    # Enclose the path with double quotes and start the path with hdfs://.
    path: "hdfs://namenode_ip:port/path/to/course.csv"

    # If the CSV file has no header, use [_c0, _c1, _c2, ..., _cn] to
    # represent its header and to indicate columns as the source of the property values.
    fields: [_c0, _c1]
    # If the CSV file has a header, use the actual column names.

    # Specifies property names defined in Nebula Graph.
    # fields for the CSV file and nebula.fields for Nebula Graph must
    # have the one-to-one correspondence relationship.
    nebula.fields: [courseId, courseName]

    # Since from Exchange 1.2.1, csv.fields is available.
    # If csv.fields is specified, fields must have the same values as csv.fields.
    # csv.fields: [courseId, courseName]

    # Specifies a column as the source of VIDs.
    # The value of vertex.field must be one column of the CSV file.
    # If the values are not of the int type, use vertex.policy to
    # set the mapping policy. "hash" is preferred.
    vertex: {
      field: _c1,
      policy: "hash"
    }

    # Specifies the separator. The default value is commas.
    separator: ","

    # If the CSV file has a header, set header to true.
    # If the CSV file has no header, set header to false (default value).
    header: false

    # Specifies the maximum number of vertex data to be written into
    # Nebula Graph in a single batch.
    batch: 256

    # Specifies the partition number of Spark.
    partition: 32

    # For the isImplicit information, refer to the application.conf file in
    # the nebula-java/tools/exchange/target/classes directory.
    isImplicit: true
  }

  # Sets for the user tag
  {
    name: user
    type: {
      source: csv
      sink: client
    }
    path: "hdfs://namenode_ip:port/path/to/user.csv"

    # Since from Exchange 1.2.1, csv.fields is available.
    # If csv.fields is used, Exchange uses the values of csv.fields as
    # the header of the CSV file, and fields must have the same values as csv.fields.
    # fields for the CSV file and nebula.fields for Nebula Graph must
```

```
      # have the one-to-one correspondence relationship.
      fields: [userId]

      # Specifies property names defined in Nebula Graph.
      # fields for the CSV file and nebula.fields for Nebula Graph must
      # have the one-to-one correspondence relationship.
      nebula.fields: [userId]

      # If csv.fields is set, its value is used to represent the header of
      # the CSV file, even though the CSV file has its own header.
      # fields and csv.fields must have the same value.
      # The value of vertex must be one of the values of csv.fields.
      csv.fields: [userId]

      # The value of vertex.field must be one column of the CSV file.
      vertex: userId
      separator: ","
      header: false
      batch: 256
      partition: 32

      # For the isImplicit information, refer to the application.conf file in
      # the nebula-java/tools/exchange/target/classes directory.
      isImplicit: true
    }

    # If more tags are necessary, refer to the preceding configuration to add more.
  ]
  # Process edges
  edges: [
    # Sets for the action edge type
    {
      # Specifies an edge type name defined in Nebula Graph
      name: action
      type: {
        # Specifies the data source. csv is used.
        source: csv

        # Specifies how to import vertex data into Nebula Graph: client or sst.
        # For more information about importing sst files, see Import SST files (doc_to_do).
        sink: client
      }

      # Specifies the HDFS path of the CSV file.
      # Enclose the path with double quotes and start the path with hdfs://.
      path: "hdfs://namenode_ip:port/path/to/actions.csv"

      # If the CSV file has no header, use [_c0, _c1, _c2, ..., _cn] to
      # represent its header and to indicate columns as the source of the property values.
      fields: [_c0, _c3, _c4, _c5, _c6, _c7, _c8]
      # If the CSV file has a header, use the actual column names.

      # Specifies property names defined in Nebula Graph.
      # fields for the CSV file and nebula.fields for Nebula Graph must
      # have the one-to-one correspondence relationship.
      nebula.fields: [actionId, duration, feature0, feature1, feature2, feature3, label]

      # Since from Exchange 1.2.1, csv.fields is available.
      # If csv.fields is used, Exchange uses the values of csv.fields as
      # the header of the CSV file and fields must have the same values as csv.fields.
      # csv.fields: [actionId, duration, feature0, feature1, feature2, feature3, label]

      # Specifies the columns as the source of the IDs of the source and target vertices.
      # If the values are not of the int type, use vertex.policy to set the mapping policy. "hash" is preferred.
      source: _c1
      target: {
        field: _c2
        policy: "hash"
      }

      # Specifies the separator. The default value is commas.
      separator: ","

      # If the CSV file has a header, set header to true.
      # If the CSV file has no header, set header to false (default value).
      header: false

      # Specifies the maximum number of vertex data to be written into
      # Nebula Graph in a single batch.
      batch: 256

      # Specifies the partition number of Spark.
      partition: 32

      # For the isImplicit information, refer to the application.conf file
      # in the nebula-java/tools/exchange/target/classes directory.
      isImplicit: true
    }
  ]
  # If more edge types are necessary, refer to the preceding configuration to add more.
}
```

Step 4. (Optional) Verify the configuration

After the configuration, run the import command with the `-D` parameter to verify the configuration file. For more information about the parameters, see Import command parameters.

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/
csv_application.conf -D
```

Step 5. Import data into Nebula Graph

When the configuration is ready, run this command to import data from CSV files into Nebula Graph. For more information about the parameters, see Import command parameters.

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/
csv_application.conf
```

Step 6. (Optional) Verify data in Nebula Graph

You can use a Nebula Graph client, such as Nebula Graph Studio, to verify the imported data. For example, in Nebula Graph Studio, run this statement.

```
GO FROM 1 OVER action;
```

If the queried destination vertices return, the data are imported into Nebula Graph.

You can use db_dump to count the data. For more information, see Dump Tool.

Step 7. (Optional) Create and rebuild indexes in Nebula Graph

After the data is imported, you can create and rebuild indexes in Nebula Graph. For more information, see nGQL User Guide.

Last update: April 15, 2021

Import data from JSON files

This article uses an example to show how to use Exchange to import data from JSON files stored on HDFS into Nebula Graph.

DATASET

The JSON file (test.json) used in this example is like `{"source":int, "target":int, "likeness":double}`, representing a `like` relationship between `source` and `target`. 21,645 records in total.

Here are some sample data:

```
{"source":53802643,"target":87847387,"likeness":0.34}
{"source":29509860,"target":57501950,"likeness":0.40}
{"source":97319348,"target":50240344,"likeness":0.77}
{"source":94295709,"target":8189720,"likeness":0.82}
{"source":78707720,"target":53874070,"likeness":0.98}
{"source":23399562,"target":20136097,"likeness":0.47}
```

ENVIRONMENT

The practice is done in macOS. Here is the environment information:

- Hardware specifications:
  - CPU: 1.7 GHz Quad-Core Intel Core i7
  - Memory: 16 GB
- Spark 2.3.0, deployed in the Standalone mode
- Hadoop 2.9.2, deployed in the Pseudo-Distributed mode
- Nebula Graph v1.2.1, deployed with Docker Compose. For more information, see Deploy Nebula Graph with Docker Compose.

PREREQUISITES

To import data from JSON files on HDFS with Exchange v1.x, do a check of these:

- Exchange v1.x is compiled. For more information, see Compile Exchange v1.x. Exchange 1.2.1 is used in this example.
- Spark is installed.
- Hadoop is installed and started.
- Nebula Graph is deployed and started. Get the information:
  - IP addresses and ports of the Graph Service and the Meta Service.
  - A Nebula Graph account with the privilege of writing data and its password.
- Get the necessary information for schema creation in Nebula Graph, including tags and edge types.

PROCEDURE

Step 1. Create a schema in Nebula Graph

Analyze the data in the JSON files and follow these steps to create a schema in Nebula Graph:

1. Confirm the essential elements of the schema.

Elements	Names	Properties
Tag	`source`	`srcId int`
Tag	`target`	`dstId int`
Edge Type	`like`	`likeness double`

2. In Nebula Graph, create a graph space named **json** and create a schema.

```
-- Create a graph space named json
CREATE SPACE json (partition_num=10, replica_factor=1);

-- Choose the json graph space
USE json;

-- Create the source tag
CREATE TAG source (srcId int);

-- Create the target tag
CREATE TAG target (dstId int);

-- Create the like edge type
CREATE EDGE like (likeness double);
```

For more information, see Quick Start of Nebula Graph Database User Guide.

Step 2. Prepare JSON files

Create separate JSON files for vertex and edge data. Store the JSON files in HDFS and get the HDFS path of the files.

> **NOTE**: In this example, only one JSON file is used to import vertex and edge data at the same time. Some vertex data representing source and target are duplicate. Therefore, during the import process, these vertices are written repeatedly. In Nebula Graph, data is overwritten when repeated insertion occurs, and the last write is read out. In practice, to increase the write speed, creating separate files for vertex and edge data is recommended.

Step 3. Edit configuration file

After compiling of Exchange, copy the `target/classes/application.conf` file and edit the configuration for JSON files. In this example, a new configuration file is named `json_ application.conf`. In this file, the vertex and edge related configuration is introduced as comments and all the items that are not used in this example are commented out. For more information about the Spark and Nebula related parameters, see Spark related parameters and Nebula Graph related parameters.

```
{
  # Spark related configuration
  spark: {
    app: {
      name: Spark Writer
    }

    driver: {
      cores: 1
      maxResultSize: 1G
    }

    cores {
      max: 16
    }
  }

  # Nebula Graph related configuration
  nebula: {
    address:{
      # Specifies the IP addresses and ports of the Graph Service and the Meta Service of Nebula Graph.
      # If multiple servers are used, separate the addresses with commas.
      # Format: "ip1:port","ip2:port","ip3:port"
      graph:["127.0.0.1:3699"]
      meta:["127.0.0.1:45500"]
    }
    # Specifies an account that has the WriteData privilege in Nebula Graph and its password.
    user: user
    pswd: password
```

```
  # Specifies a graph space name
  space: json

  connection {
    timeout: 3000
    retry: 3
  }

  execution {
    retry: 3
  }

  error: {
    max: 32
    output: /tmp/errors
  }

  rate: {
    limit: 1024
    timeout: 1000
  }
}

# Process vertices
tags: [
  # Sets for the source tag
  {
    # Specifies a tag name defined in Nebula Graph
    name: source
    type: {
      # Specifies the data source. json is used.
      source: json

      # Specifies how to import vertex data into Nebula Graph: client or sst.
      # For more information about importing sst files, see Import SST files (doc to do).
      sink: client
    }

    # Specifies the HDFS path of the JSON file.
    # Enclose the path with double quotes and start the path with hdfs://.
    path: "hdfs://namenode_ip:port/path/to/test.json"

    # Specifies the keys in the JSON file.
    # Their values are used as the source of the srcId property
    # defined in Nebula Graph.
    # If more than one key is specified, separate them with commas.
    fields: ["source"]
    nebula.fields: ["srcId"]

    # Specifies the values of a key in the JSON file as
    # the source of the VID in Nebula Graph.
    # If the values are not of the int type, use vertex.policy to
    # set the mapping policy. "hash" is preferred.
    #
    # vertex: {
    #   field: key_name_in_json
    #   policy: "hash"
    # }
    vertex: source

    batch: 256
    partition: 32

    # For the isImplicit information, refer to the application.conf file
    # in the nebula-java/tools/exchange/target/classes directory.
    isImplicit: true
  }
  # Sets for the target tag
  {
    name: target
    type: {
      source: json
      sink: client
    }
    path: "hdfs://namenode_ip:port/path/to/test.json"
    fields: ["target"]
    nebula.fields: ["dstId"]
    vertex: "target"
    batch: 256
    partition: 32
    isImplicit: true
  }
  # If more tags are necessary, refer to the preceding configuration to add more.
]

# Process edges
edges: [
  # Sets for the like edge type
  {
    # Specifies an edge type name defined in Nebula Graph
    name: like
    type: {
      # Specifies the data source. json is used.
```

```
        source: json

        # Specifies how to import vertex data into Nebula Graph: client or sst.
        # For more information about importing sst files, see Import SST files (doc to do).
        sink: client
      }

      # Specifies the HDFS path of the JSON file.
      # Enclose the path with double quotes and start the path with hdfs://.
      path: "hdfs://namenode_ip:port/path/to/test.json"

      # Specifies the keys in the JSON file.
      # Their values are used as the source of the likeness property defined in Nebula Graph.
      # If more than one key is specified, separate them with commas.
      fields: ["likeness"]
      nebula.fields: ["likeness"]

      # Specifies the values of two keys in the JSON file as the source
      # of the IDs of source and destination vertices of the like edges in Nebula Graph.
      # If the values are not of the int type, use source.policy and/or
      # target.policy to set the mapping policy. "hash" is preferred.
      # source: {
      #   field: key_name_in_json
      #   policy: "hash"
      # }
      # target: {
      #   field: key_name_in_json
      #   policy: "hash"
      # }
      source: "source"
      target: "target"

      batch: 256
      partition: 32
      isImplicit: true
    }
  # If more edge types are necessary, refer to the preceding configuration to add more.
  ]
}
```

Step 4. (Optional) Verify the configuration

After the configuration, run the import command with the `-D` parameter to verify the configuration file. For more information about the parameters, see Import command parameters.

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/
json_application.conf -D
```

Step 5. Import data into Nebula Graph

When the configuration is ready, run this command to import data from JSON files into Nebula Graph. For more information about the parameters, see Import command parameters.

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/
json_application.conf
```

Step 6. (Optional) Verify data in Nebula Graph

You can use a Nebula Graph client, such as Nebula Graph Studio, to verify the imported data. For example, in Nebula Graph Studio, run this statement.

```
GO FROM 53802643 OVER like;
```

If the queried destination vertices return, the data are imported into Nebula Graph.

You can use db_dump to count the data. For more information, see Dump Tool.

Step 7. (Optional) Create and rebuild indexes in Nebula Graph

After the data is imported, you can create and rebuild indexes in Nebula Graph. For more information, see nGQL User Guide.

Last update: April 15, 2021

Import data from HIVE

This article uses an example to show how to use Exchange to import data from HIVE into Nebula Graph.

DATASET

In this article, the Social Network: MOOC User Action Dataset provided by Stanford Network Analysis Platform (SNAP) and 97 unique course names obtained from the public network are used as the sample dataset. The dataset contains:

- Two vertex types (`user` and `course`), 7,144 vertices in total.
- One edge type (`action`), 411,749 edges in total.

You can download the example dataset from the nebula-web-docker repository.

In this example, the dataset is stored in a database named `mooc` in HIVE, and the information of all vertices and edges is stored in the `users`, `courses`, and `actions` tables. Here are the structures of all the tables.

```
scala> sql("describe mooc.users").show
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|  userid|   bigint|   null|
+--------+---------+-------+

scala> sql("describe mooc.courses").show
+----------+---------+-------+
|  col_name|data_type|comment|
+----------+---------+-------+
|  courseid|   bigint|   null|
|coursename|   string|   null|
+----------+---------+-------+

scala> sql("describe mooc.actions").show
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|actionid|   bigint|   null|
|   srcid|   bigint|   null|
|   dstid|   string|   null|
|duration|   double|   null|
|feature0|   double|   null|
|feature1|   double|   null|
|feature2|   double|   null|
|feature3|   double|   null|
|   label|  boolean|   null|
+--------+---------+-------+
```

NOTE: `bigint` in HIVE equals to `int` in Nebula Graph.

ENVIRONMENT

The practice is done in macOS. Here is the environment information:

- Hardware specifications:
  - CPU: 1.7 GHz Quad-Core Intel Core i7
  - Memory: 16 GB
- Spark 2.4.7, deployed in the Standalone mode
- Hadoop 2.9.2, deployed in the Pseudo-Distributed mode
- HIVE 2.3.7, with MySQL 8.0.22
- Nebula Graph v1.2.1, deployed with Docker Compose. For more information, see Deploy Nebula Graph with Docker Compose.

PREREQUISITES

To import data from HIVE with Exchange v1.x, do a check of these:

- Exchange v1.x is compiled. For more information, see Compile Exchange v1.x. Exchange 1.2.1 is used in this example.
- Spark is installed.
- Hadoop is installed and started and the hive metastore database (MySQL is used in this example) is started.
- Nebula Graph is deployed and started. Get the information:
  - IP addresses and ports of the Graph Service and the Meta Service.
  - A Nebula Graph account with the privilege of writing data and its password.
- Get the necessary information for schema creation in Nebula Graph, including tags and edge types.

PROCEDURE

Step 1. Create a schema in Nebula Graph

Follow these steps to create a schema in Nebula Graph:

1. Confirm the essential elements of the schema.

Elements	Names	Properties
Tag	`user`	`userId int`
Tag	`course`	`courseId int, courseName string`
Edge Type	`action`	`actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double`

2. In Nebula Graph, create a graph space named **hive** and create a schema.

```
-- Create a graph space named hive
CREATE SPACE hive(partition_num=10, replica_factor=1);

-- Choose the hive graph space
USE hive;

-- Create the user tag
CREATE TAG user(userId int);

-- Create the course tag
CREATE TAG course(courseId int, courseName string);

-- Create the action edge type
CREATE EDGE action (actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double);
```

For more information, see Quick Start of Nebula Graph Database User Guide.

Step 2. Verify the HIVE SQL statements

When spark-shell starts, run these statements one by one to make sure that Spark can read data from HIVE.

```
scala> sql("select userid from mooc.users").show
scala> sql("select courseid, coursename from mooc.courses").show
scala> sql("select actionid, srcid, dstid, duration, feature0, feature1, feature2, feature3, label from mooc.actions").show
```

Here is an example of data read from the `mooc.actions` table.

```
+--------+-----+--------------------+--------+------------+------------+-----------+-----------+-----+
|actionid|srcid|               dstid|duration|    feature0|    feature1|   feature2|   feature3|label|
+--------+-----+--------------------+--------+------------+------------+-----------+-----------+-----+
|       0|    0|Environmental Dis...|     0.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       1|    0|  History of Ecology|     6.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       2|    0|      Women in Islam|    41.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       3|    0|  History of Ecology|    49.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       4|    0|      Women in Islam|    51.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       5|    0|Legacies of the A...|    55.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       6|    0|          ITP Core 2|    59.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       7|    0|The Research Pape...|    62.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       8|    0|        Neurobiology|    65.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       9|    0|           Wikipedia|   113.0|-0.319991479|-0.435701433|1.108826104|12.77723482|false|
|      10|    0|Media History and...|   226.0|-0.319991479|-0.435701433|0.607804941|149.4512115|false|
```

```
|      11|    0|             WIKISOO|   974.0|-0.319991479|-0.435701433|1.108826104|3.344522776|false|
|      12|    0|Environmental Dis...|  1000.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      13|    0|             WIKISOO|  1172.0|-0.319991479|-0.435701433|1.108826104|1.136866766|false|
|      14|    0|      Women in Islam|  1182.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      15|    0|  History of Ecology|  1185.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      16|    0|Human Development...|  1687.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      17|    1|Human Development...|  7262.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      18|    1|  History of Ecology|  7266.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      19|    1|      Women in Islam|  7273.0|-0.319991479|-0.435701433|0.607804941|0.936170765|false|
+--------+-----+--------------------+--------+------------+------------+-----------+-----------+-----+
only showing top 20 rows
```

Step 3. Edit configuration file

After compiling of Exchange, copy the `target/classes/application.conf` file and edit the configuration for HIVE. In this example, a new configuration file is named `hive_ application.conf`. In this file, the vertex and edge related configuration is introduced as comments and all the items that are not used in this example are commented out. For more information about the Spark and Nebula related parameters, see Spark related parameters and Nebula Graph related parameters.

```
{
  # Spark related configuration
  spark: {
    app: {
      name: Spark Writer
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores {
      max: 16
    }
  }
  # Nebula Graph related configuration
  nebula: {
    address:{
      # Specifies the IP addresses and ports of the Graph Service and the Meta Service of Nebula Graph
      # If multiple servers are used, separate the addresses with commas,
      # "ip1:port","ip2:port","ip3:port"
      graph:["127.0.0.1:3699"]
      meta:["127.0.0.1:45500"]
    }

    # Specifies an account that has the WriteData privilege in Nebula Graph and its password
    user: user
    pswd: password

    # Specifies a graph space name
    space: hive
    connection {
      timeout: 3000
      retry: 3
    }
    execution {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }

  # Process vertices
  tags: [
    # Sets for the user tag
    {
      # Specifies a tag name defined in Nebula Graph
      name: user
      type: {
        # Specifies the data source. hive is used.
        source: hive
        # Specifies how to import vertex data into Nebula Graph: client or sst.
        # For more information about importing sst files, see Import SST files (doc to do).
        sink: client
      }

      # Specifies the SQL statement to read data from the users table in the mooc database
      exec: "select userid from mooc.users"

      # Specifies the column names from the users table to fields.
      # Their values are used as the source of the userId (nebula.fields) property defined in Nebula Graph.
      # If more than one column name is specified, separate them with commas.
      # fields for the HIVE and nebula.fields for Nebula Graph must have the one-to-one correspondence relationship.
      fields: [userid]
      nebula.fields: [userId]
```

```
      # Specifies a column as the source of VIDs.
      # The value of vertex must be one column name in the exec sentence.
      # If the values are not of the int type, use vertex.policy to
      # set the mapping policy. "hash" is preferred.
      # Refer to the configuration of the course tag.
      vertex: userid

      # Specifies the maximum number of vertex data to be written into
      # Nebula Graph in a single batch.
      batch: 256

      # Specifies the partition number of Spark.
      partition: 32

      # For the isImplicit information, refer to the application.conf file in
      # the nebula-java/tools/exchange/target/classes directory.
      isImplicit: true
    }

    # Sets for the course tag
    {
      name: course
      type: {
        source: hive
        sink: client
      }
      exec: "select courseid, coursename from mooc.courses"
      fields: [courseid, coursename]
      nebula.fields: [courseId, courseName]

      # Specifies a column as the source of VIDs.
      # The value of vertex.field must be one column name in the exec sentence.
      # If the values are not of the int type, use vertex.policy to
      # set the mapping policy. "hash" is preferred.
      vertex: {
        field: coursename
        policy: "hash"
      }
      batch: 256
      partition: 32
      isImplicit: true
    }
    # If more tags are necessary, refer to the preceding configuration to add more.
  ]

  # Process edges
  edges: [
    # Sets for the action edge type
    {
      # Specifies an edge type name defined in Nebula Graph
      name: action

      type: {
        # Specifies the data source. hive is used.
        source: hive

        # Specifies how to import vertex data into Nebula Graph: client or sst
        # For more information about importing sst files,
        # see Import SST files (doc to do).
        sink: client
      }

      # Specifies the SQL statement to read data from the actions table in
      # the mooc database.
      exec: "select actionid, srcid, dstid, duration, feature0, feature1, feature2, feature3, label from mooc.actions"

      # Specifies the column names from the actions table to fields.
      # Their values are used as the source of the properties of
      # the action edge type defined in Nebula Graph.
      # If more than one column name is specified, separate them with commas.
      # fields for the HIVE and nebula.fields for Nebula Graph must
      # have the one-to-one correspondence relationship.
      fields: [actionid, duration, feature0, feature1, feature2, feature3, label]
      nebula.fields: [actionId, duration, feature0, feature1, feature2, feature3, label]

      # source specifies a column as the source of the IDs of
      # the source vertex of an edge.
      # target specifies a column as the source of the IDs of
      # the target vertex of an edge.
      # The value of source.field and target.field must be
      # column names set in the exec sentence.
      # If the values are not of the int type, use vertex.policy to
      # set the mapping policy. "hash" is preferred.
      source: srcid
      target: {
        field: dstid
        policy: "hash"
      }

      # Specifies the maximum number of vertex data to be
      # written into Nebula Graph in a single batch.
      batch: 256

      # Specifies the partition number of Spark.
```

```
      partition: 32
    }
  ]
}
```

Step 4. (Optional) Verify the configuration

After the configuration, run the import command with the `-D` parameter to verify the configuration file. For more information about the parameters, see Import command parameters.

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/
hive_application.conf -h -D
```

Step 5. Import data into Nebula Graph

When the configuration is ready, run this command to import data from HIVE into Nebula Graph. For more information about the parameters, see Import command parameters.

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/
hive_application.conf -h
```

Step 6. (Optional) Verify data in Nebula Graph

You can use a Nebula Graph client, such as Nebula Graph Studio, to verify the imported data. For example, in Nebula Graph Studio, run this statement.

```
GO FROM 1 OVER action;
```

If the queried destination vertices return, the data are imported into Nebula Graph.

You can use db_dump to count the data. For more information, see Dump Tool.

Step 7. (Optional) Create and rebuild indexes in Nebula Graph

After the data is imported, you can create and rebuild indexes in Nebula Graph. For more information, see nGQL User Guide.

Last update: April 15, 2021

4.1.4 Parameter reference

Spark related parameters

To import data, you must set parameters for Spark. This table lists some generally-used parameters. For more Spark-related parameters, see Apache Spark documentation. For more information, see the examples.

Parameters	Default	Data type	Required?	Description
`spark.app.name`	Spark Writer	`string`	No	Specifies the name of the Spark Driver Program.
`spark.driver.cores`	1	`int`	No	Specifies the number of cores to use for the driver process, only in cluster mode.
`spark.driver.maxResultSize`	1G	`string`	No	Specifies the limit of the total size of serialized results of all partitions for each Spark action (e.g. collect) in bytes. Should be at least 1M, or 0 for unlimited.
`spark.cores.max`	None	`int`	No	When the driver program runs on a standalone deployed cluster or a Mesos cluster in "coarse-grained" sharing mode, the maximum amount of CPU cores to request for the application from across the cluster (not from each machine). If not set, the default will be `spark.deploy.defaultCores` on the standalone cluster manager of Spark, or infinite (all available cores) on Mesos.

Last update: April 8, 2021

Nebula Graph related parameters

To import data, you must set parameters for Nebula Graph. This table lists all the Nebula Graph related parameters. For more information, see the examples.

Parameters	Default	Data Type	Required?	Description
`nebula.address.graph`	None	`list[string]`	Yes	Specifies the addresses and ports used by the Graph Service of Nebula Graph. Multiple addresses must be separated with commas. In the format of `"ip1:port","ip2:port","ip3:port"`.
`nebula.address.meta`	None	`list[string]`	Yes	Specifies the addresses and ports used by the Meta Service of Nebula Graph. Multiple addresses must be separated with commas. In the format of `"ip1:port","ip2:port","ip3:port"`.
`nebula.user`	`user`	`string`	Yes	Specifies an account of Nebula Graph. The default value is `user`. If authentication is enabled in Nebula Graph: - If no account is created, use `root`. - If a specified account is created and given the write permission to a graph space, you can use this account.
`nebula.pswd`	`password`	`string`	Yes	Specifies the password of the specified account. The default password for the `user` account is `password`. If authentication is enabled in Nebula Graph: - For the `root` account, use `nebula`. - For another account, use the specified password.
`nebula.space`	None	`string`	Yes	Specifies the name of the graph space to import data.
`nebula.connection.timeout`	3000	`int`	No	Specifies the period of timeout for Thrift connection. Unit: ms.
`nebula.connection.retry`	3	`int`	No	Specifies the number of retries for Thrift connection.
`nebula.execution.retry`	3	`int`	No	Specifies the number of execution retries of an nGQL statements
`nebula.error.max`	32	`int`	No	Specifies the maximum number of failures during the import process. When the specified number of failures occur, the submitted Spark job stops automatically.
`nebula.error.output`	None	`string`	Yes	Specifies a logging directory on the Nebula Graph cluster for the error message.

Last update: April 8, 2021

Import command parameters

When the configuration file is ready, replace `master-node-url` and `exchange-1.x.y.jar` in this command and run it to import the data from the specified source into Nebula Graph.

```
$SPARK_HOME/bin/spark-submit  --class com.vesoft.nebula.tools.importer.Exchange --master "master-node-url" target/exchange-1.x.y.jar -c /path/to/conf/
application.conf
```

This table lists all the parameters in the preceding command.

Parameters	Required?	Default	Description
`--class`	Yes	None	Specifies the entry point of Exchange.
`--master`	Yes	None	Specifies the URL of the Master node of the specified Spark cluster. For more information, see master-urls in Spark Documentation
`-c` / `--config`	Yes	None	Specifies the path of the Exchange configuration file.
`-h` / `--hive`	No	`false`	If you want to import data from HIVE, add this parameter.
`-D` / `--dry`	No	`false`	Before data import, add this parameter to do a check of the format of the configuration file, but not the configuration of `tags` and `edges`. Do not use this parameter when you import data.

Last update: April 8, 2021

5. Nebula Graph Studio

5.1 About Nebula Graph Studio

5.1.1 What is Nebula Graph Studio

Nebula Graph Studio (Studio in short) is a browser-based visualization tool to manage Nebula Graph. It provides you with a graphical user interface to manipulate graph schemas, import data, explore graph data, and run nGQL statements to retrieve data. With Studio, you can quickly become a graph exploration expert from scratch.

Release distributions

Studio has two release distributions:

- Docker-based Studio: You can deploy Studio with Docker and connect Studio to Nebula Graph. For more information, see Deploy Studio.
- Studio on Cloud: When you created a Nebula Graph instance on Nebula Graph Cloud Service, you can connect to Studio on Cloud with one click. For more information, see Nebula Graph Cloud Service User Guide.

Both release distributions have different limitations. For more information, see Limitations.

Features

Studio provides these features:

- Graphical user interface (GUI) makes Nebula Graph management more user-friendly:
  - On the **Schema** page, you can manage schemas with a graphical user interface. It helps you quickly get started with Nebula Graph.
  - On the **Console** page, you can run nGQL statements and read the results in a human-friendly way.
  - On the **Import** page, you can operate batch import of vertex and edge data with clicks, and view a real-time import log.
- On the **Explore** page, you can explore the graph data. It helps you dig the relationships among data and improves the efficiency of data analysis.

Scenarios

You can use Studio in one of these scenarios:

- You have a dataset, and you want to explore and analyze data in a visualized way. You can use Docker Compose or Nebula Graph Cloud Service to deploy Nebula Graph and then use Studio to explore or analyze data in a visualized way.
- You have deployed Nebula Graph and imported a dataset. You want to use a GUI to run nGQL statements or explore and analyze graph data in a visualized way.
- You are a beginner of nGQL (Nebula Graph Query Language) and you prefer to use a GUI rather than a command-line interface (CLI) to learn the language.

Authentication

For Studio on Cloud, only the instance creator and the Nebula Graph Cloud Service accounts that are authorized to manipulate data in Nebula Graph can connect to Studio. For more information, see Nebula Graph Cloud Service User Guide.

For Docker-based Studio, authentication is not enabled in Nebula Graph by default and you can sign in to Studio with the default account and password (`user` and `password`). When authentication is enabled, you must sign in to Studio with the assigned account and password.

For more information about authentication, see Nebula Graph Database Manual.

Last update: April 8, 2021

5.1.2 Limitations

This article introduces the limitations on Studio.

Nebula Graph versions

For now, only Nebula Graph v1.2.1 and earlier versions can be used with Studio. Nebula Graph v2.0.0-alpha is not supported.

Architecture

For now, Docker-based Studio supports x86_64 architecture only.

Upload data

During the public beta of Nebula Graph Cloud Service, Studio on Cloud has these limitations:

- Only CSV files without headers are supported, and only commas are acceptable separator.
- Each file of a maximum of 100 MB is supported.
- A total amount of a maximum of 1 GB is supported for each Nebula graph instance.
- Each file is stored for only one calendar day.

On Docker-based Studio, only CSV files without headers can be uploaded, but no limitations are applied to the size and store period for a single file. The maximum data volume depends on the storage capacity of your machine.

Data backup

For now, you can export the queried results in the CSV format on the **Console** page. No other backup methods are available.

nGQL statements

On the **Console** page of Docker-based Studio, all the nGQL syntaxes except these are supported:

- `USE <space_name>`: You cannot run such a statement on the **Console** page to select a graph space. As an alternative, you can click a graph space name in the drop-down list of **Current Graph Space**.
- You cannot use line breaks (\). As an alternative, you can use the Enter key to split a line.

For Studio on Cloud, besides the preceding syntax, you cannot run these account and role management statements on the **Console** page:

- `CREATE USER`
- `ALTER USER`
- `CHANGE PASSWORD`
- `DROP USER`
- `GRANT ROLE`
- `REVOKE ROLE`

For more information about the preceding statements, see Nebula Graph Database Manual.

Browser

We recommend that you use Chrome to get access to Studio.

Last update: April 15, 2021

5.1.3 Check updates

Studio is in development. To get updated with its development, visit GitHub and read its Changelog

Studio on Cloud

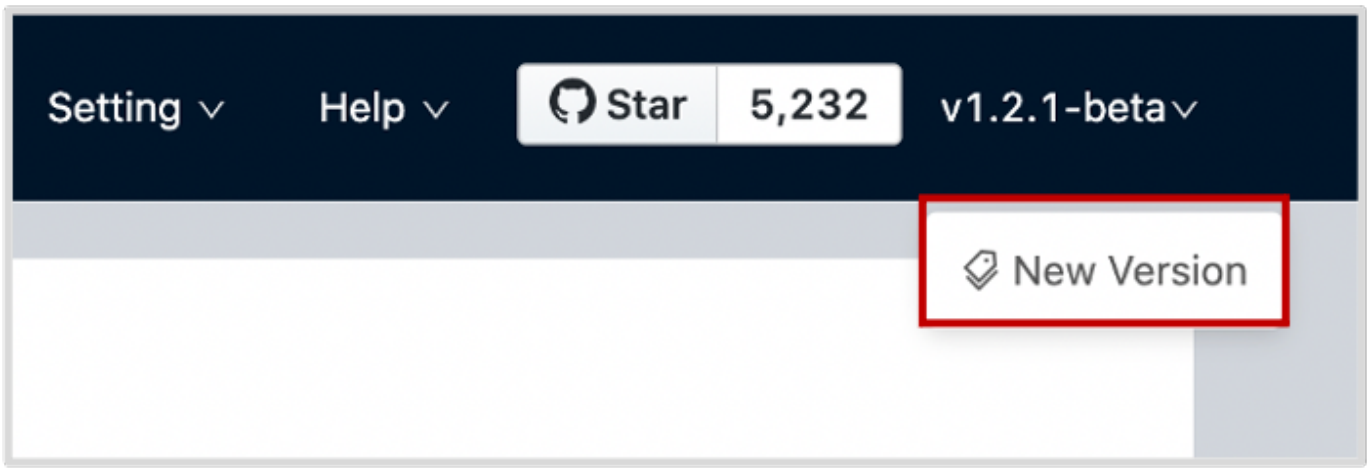
Studio on Cloud is deployed automatically when you create a Nebula Graph instance on Nebula Graph Cloud Service. You cannot update its version. During the public beta, v1.1.1-beta is deployed.

Docker-based Studio

For Docker-based Studio, we recommend that you run this command in the `nebula-web-docker` directory to update the Docker image and start the service:

```
docker-compose pull && docker-compose up -d
```

To view the changelog, on the upper-right corner of the page, click the version and then `New version`.



Last update: April 8, 2021

5.2 Deploy and connect

5.2.1 Deploy Studio

Studio on Cloud can be used on Nebula Graph Cloud Service. When you create a Nebula Graph instance on Nebula Graph Cloud Service, Studio on Cloud is deployed automatically. For more information, see Nebula Graph Cloud Service User Guide. For Docker-based Studio, you must deploy it. This article introduces how to deploy Docker-based Studio.

Prerequisites

Before you deploy Docker-based Studio, you must do a check of these:

- The Nebula Graph services are deployed and started. For more information, see Nebula Graph Database Manual.

  NOTE: Different methods are available for you to deploy Nebula Graph. If this is your first time to use Nebula Graph, we recommend that you use Docker Compose to deploy Nebula Graph. For more information, see Deploy Nebula Graph with Docker Compose.

- On the machine where Studio will run, Docker Compose is installed and started. For more information, see Docker Compose Documentation.

Procedure

To deploy and start Docker-based Studio, run these commands one by one:

1. Download the configuration files for the deployment.

```
git clone https://github.com/vesoft-inc/nebula-web-docker.git
```

2. Change to the `nebula-web-docker` directory.

```
cd nebula-web-docker
```

3. Pull the Docker image of Studio.

```
docker-compose pull
```

4. Build and start Docker-based Studio. In this command, `-d` is to run the containers in the background.

```
docker-compose up -d
```

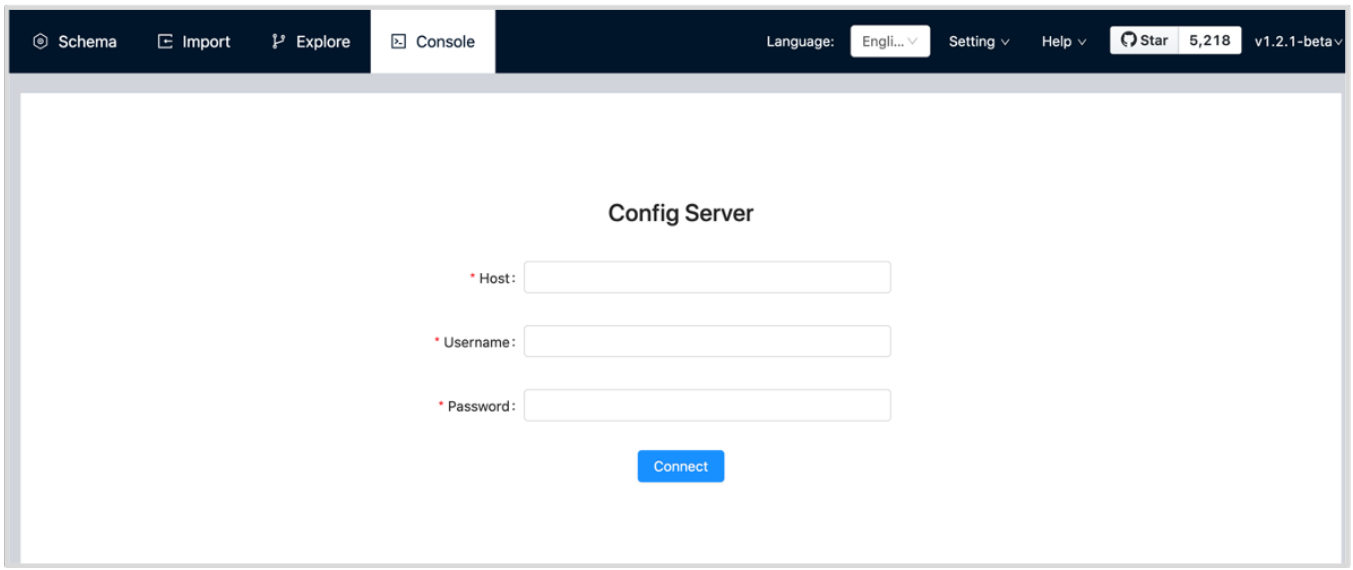
If these lines return, Docker-based Studio is deployed and started.

```
Creating docker_importer_1 ... done
Creating docker_client_1   ... done
Creating docker_web_1      ... done
Creating docker_nginx_1    ... done
```

5. When Docker-based Studio is started, use `http://ip address:7001` to get access to Studio.

   NOTE: Run `ifconfig` or `ipconfig` to get the IP address of the machine where Docker-based Studio is running. On the machine running Docker-based Studio, you can use `http://localhost:7001` to get access to Studio.

If you can see the **Config Server** page on the browser, Docker-based Studio is started successfully.



Next to do

On the **Config Server** page, connect Docker-based Studio to Nebula Graph. For more information, see Connect to Nebula Graph.

Last update: April 8, 2021

5.2.2 Connect to Nebula Graph

On Nebula Graph Cloud Service, after a Nebula Graph instance is created, Studio on Cloud is deployed automatically, and you can connect to Studio with one click. For more information, see Nebula Graph Cloud Service User Guide. But for Docker-based Studio, when it is started, you must configure it to connect to Nebula Graph. This article introduces how to connect Docker-based Studio to Nebula Graph.

Prerequisites

Before you connect Docker-based Studio to Nebula Graph, you must do a check of these:

- The Nebula Graph services and Studio are started. For more information, see Deploy Studio.
- You have the IP address and the port used by the Graph service of Nebula Graph. The default port is `3699`.

  NOTE: Run `ifconfig` or `ipconfig` on the machine to get the IP address.
- You have a Nebula Graph account and its password.

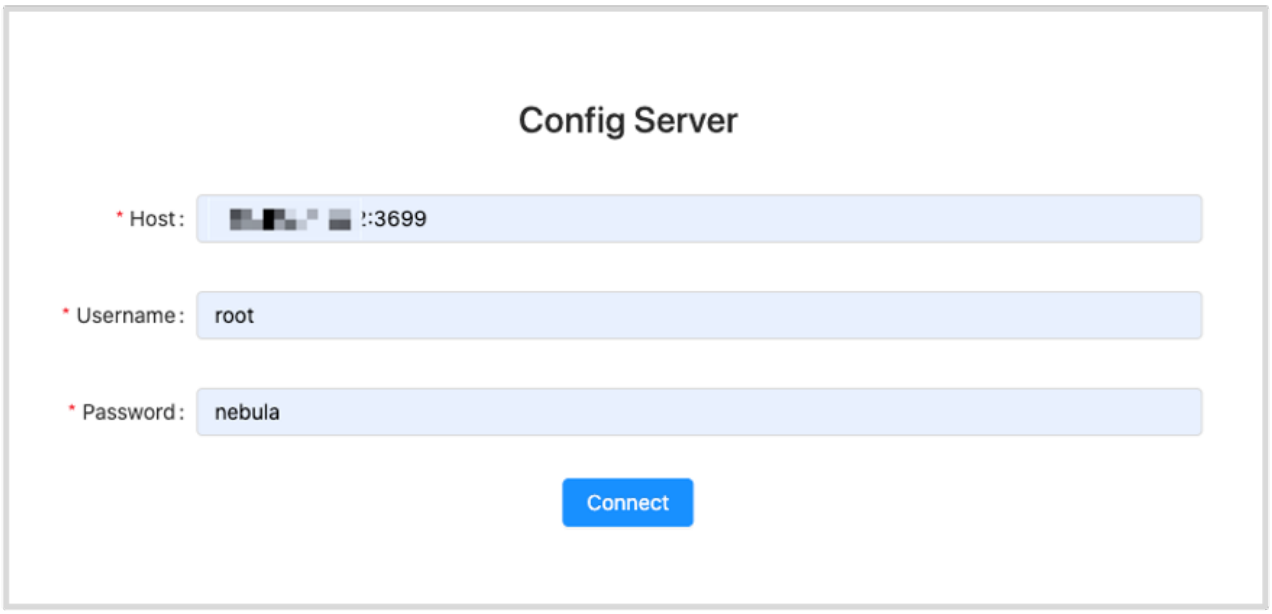
  NOTE: If authentication is enabled in Nebula Graph and different role-based accounts are created, you must use the assigned account to connect to Nebula Graph. If authentication is disabled, you can use the default username (`user`) and the default password (`password`) to connect to Nebula Graph. For more information, see Nebula Graph Database Manual.

Procedure

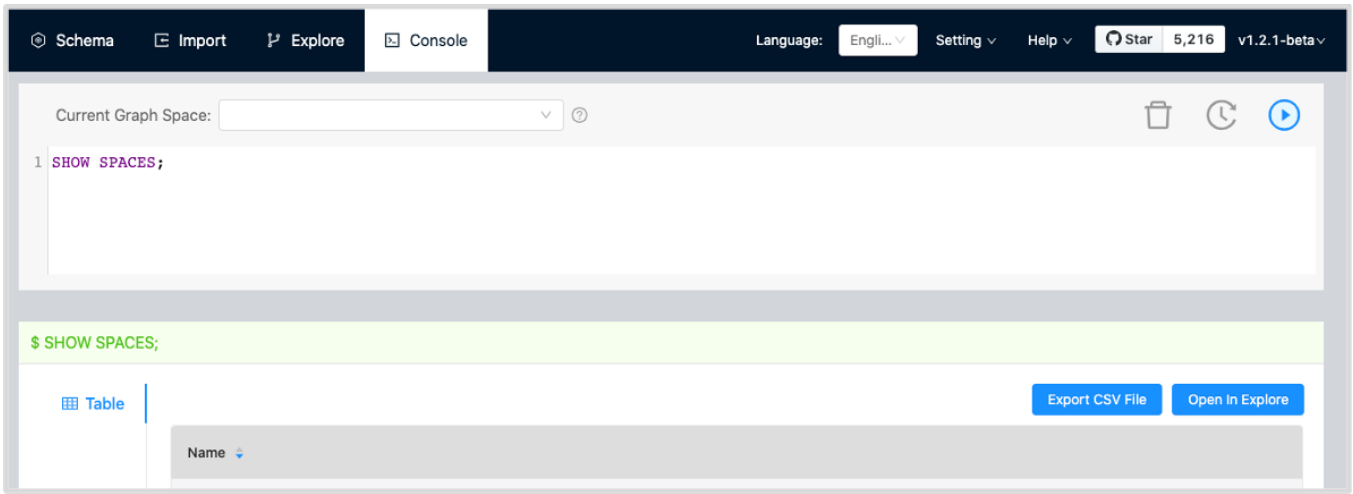
To connect Docker-based Studio to Nebula Graph, follow these steps:

1. On the **Config Server** page of Studio, configure these fields:
   - **Host**: Enter the IP address and the port of the Graph service of Nebula Graph. The valid format is `IP:port`. The default port is `3699`.

     NOTE: When Nebula Graph and Studio are deployed on the same machine, you must enter the IP address of the machine, but not `127.0.0.1` or `localhost`, in the **Host** field.
   - **Username** and **Password**: Enter a valid Nebula Graph account and its password.
     - If authentication is not enabled, you can use `user` and `password`.
     - If authentication is enabled and no accounts are created, you must use `root` and its password `nebula`.
     - If authentication is enabled and different role-based accounts are created, you must use the assigned account and its password.



2. After the configuration, click the **Connect** button.
   If you can see the **Console** page, Docker-based Studio is successfully connected to Nebula Graph.



One session continues up to 30 minutes. If you do not operate Studio within 30 minutes, the active session will time out and you must connect to Nebula Graph again.

Next to do

When Studio is successfully connected to Nebula Graph, you can do these operations:

- If your account has GOD or ADMIN privilege, you can create a schema on the **Console** page or on the **Schema** page.
- If your account has GOD, ADMIN, DBA, or USER privilege, you can batch import data on the **Import** page or insert data with nGQL statements on the **Console** page.
- If your account has GOD, ADMIN, DBA, USER, or GUEST privilege, you can retrieve data with nGQL statements on the **Console** page or explore and analyze data on the **Explore** page.

Last update: April 8, 2021

5.3 Quick start

5.3.1 Design a schema

To operate graph data in Nebula Graph with Studio, you must have a graph schema. This article introduces how to design a graph schema for Nebula Graph.

A graph schema for Nebula Graph must have these essential elements:

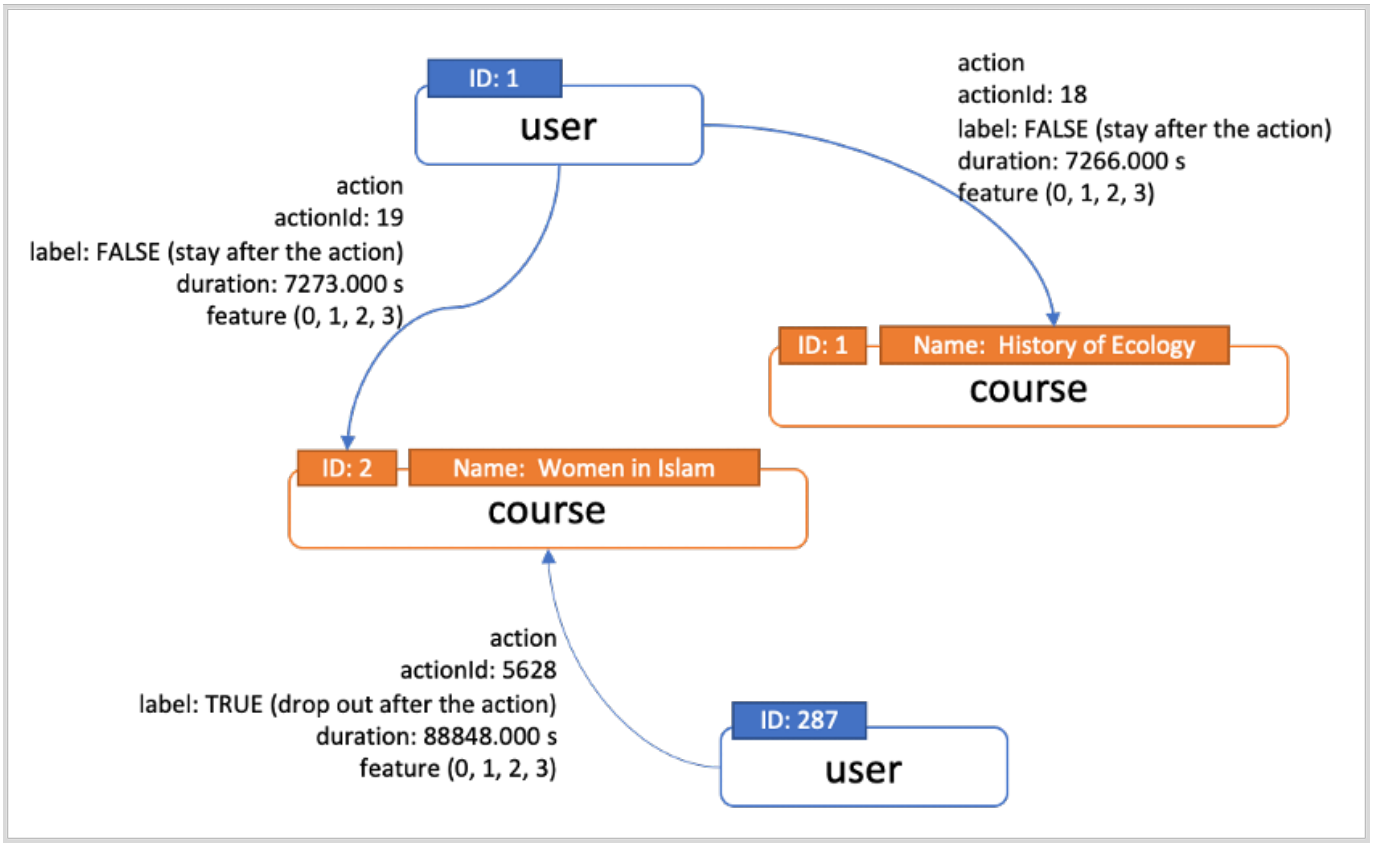
- Tags (namely vertex types) and their properties.
- Edge types and their properties

In this article, the Social Network: MOOC User Action Dataset and 97 distinct course names are used to introduce how to design a schema.

This table gives all the essential elements of the schema.

Element	Name	Property name (Data type)	Description
Tag	**user**	`userId` (`int`). The `userId` values are used to generate VIDs of user vertices.	Represents users of the specified MOOC platform.
Tag	**course**	`courseId` (`int`) and `courseName` (`string`). The `courseName` values are processed by the `Hash()` function to generate VIDs of course vertices. In Nebula Graph, VIDs must be distinct through a graph space. But in the source dataset, some `courseId` values are duplicate with some `userId` values, so the `courseId` values cannot be used to generate the VIDs of course vertices.	Represents the courses on the specified MOOC platform.
Edge type	**action**	- `actionId` (`int`) - `duration` (`double`): Represents the duration of an action measured in seconds from the beginning. Its values are equal to the `timestamp` values in the data source. - `label` (`bool`): Represents whether a user drops out after an action. `TRUE` indicates a drop-out action, `FALSE` otherwise. - `feature0` (`double`) - `feature1` (`double`) - `feature2` (`double`) - `feature3` (`double`)	Represents actions taken by users on the specified MOOC platform. An action links a user and a course and the direction is from a user to a course. It has four features.

This figure shows the relationship (**action**) between a **user** and a **course** on the MOOC platform.



Last update: April 8, 2021

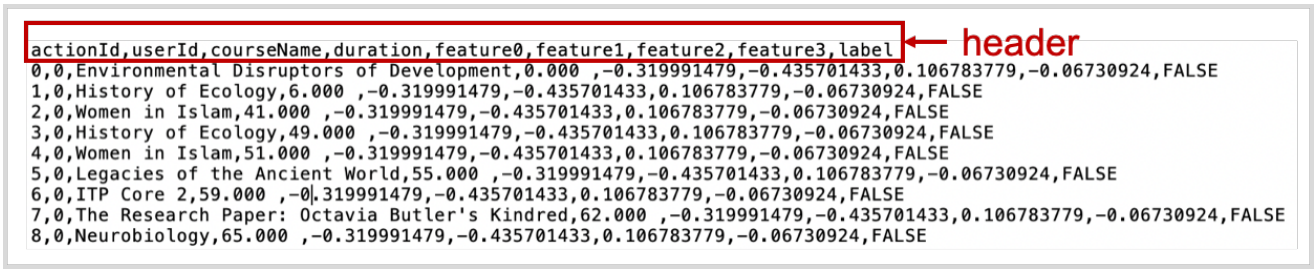
5.3.2 Prepare CSV files

With Studio, you can bulk import vertex and edge data into Nebula Graph. Currently, only CSV files without headers are supported. Each file represents vertex or edge data of one type.

To create applicable CSV files, process the source data as follows:

1. Generate CSV files for vertex and edge data. Only commas are acceptable separator.
   - `user.csv`: Contains the vertices representing users with the `userId` property.
   - `course.csv`: Contains the vertices representing courses with the `courseId` and `courseName` properties.
   - `actions.csv` contains:
     - The edges representing actions with the `actionId`, `label`, `duration`, `feature0`, `feature1`, `feature2`, and `feature3` properties. For the `label` column, 1 is replaced with `TRUE` and 0 is replaced with `FALSE`.
     - The `userId` column representing the source vertices of the edges.
     - The `courseName` column representing the destination vertices of the edges.

   This figure shows an example of a CSV file with the header.



2. Delete all the headers from the CSV files.

Last update: April 8, 2021

5.3.3 Create a schema

To bulk import data into Nebula Graph, you must have a schema. You can create a schema with the **Console** module or the **Schema** module of Studio.

> **NOTE**: You can use nebula-console to create a schema. For more information, see Deploy Nebula Graph with Docker Compose and Get started with Nebula Graph.

Prerequisites

To create a schema on Studio, you must do a check of these:

- Studio is connected to Nebula Graph.
- Your account has the privileges of GOD, ADMIN, or DBA. For more information, see Nebula Graph roles.
- The schema is designed.
- A graph space is created.

  > **NOTE**: If no graph space exists, and your account has GOD privileges, you can create a graph space on the **Console** or **Schema** page.

Create a schema with Schema

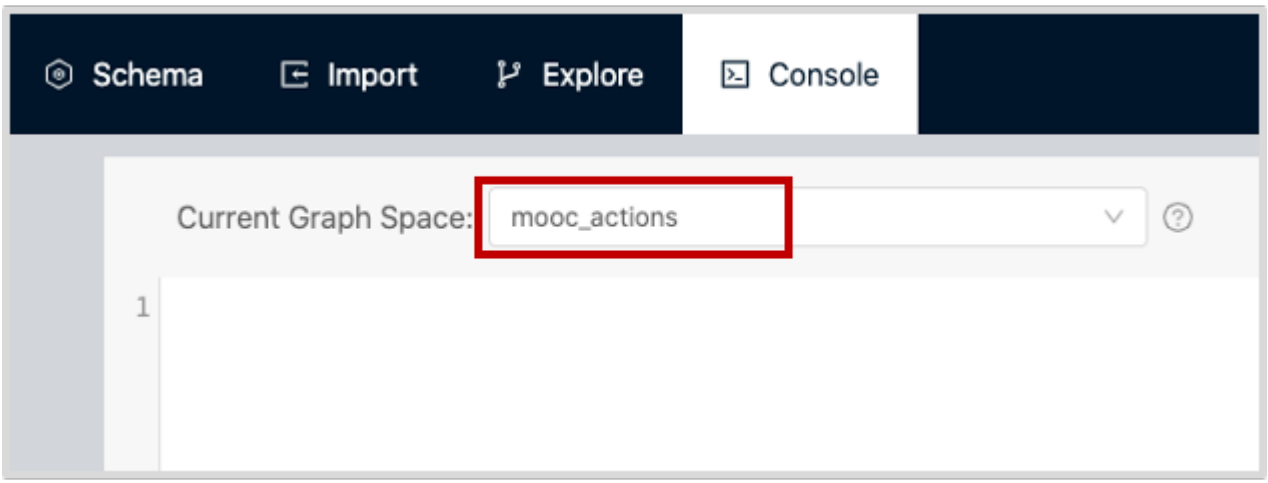
To create a schema on the **Schema** page, make sure that the version of Studio is v1.2.1-beta or later then follow these steps:

1. Create tags. For more information, see Operate tags.
2. Create edge types. For more information, see Operate edge types.

Create a schema with Console

To create a schema on the **Console** page, follow these steps:

1. In the toolbar, click the **Console** tab.
2. In the **Current Graph Space** field, choose a graph space name. In this example, **mooc_actions** is used.



3. In the input box, enter these statements one by one and click the button  .

```
-- To create a tag named "user", with one property
nebula> CREATE TAG user (userId int);

-- To create a tag named "course", with two properties
nebula> CREATE TAG course (courseId int, courseName string);

-- To create an edge type named "action", with seven properties
```

```
nebula> CREATE EDGE action (actionId int, duration double, label bool, \
feature0 double, feature1 double, feature2 double, feature3 double);
```

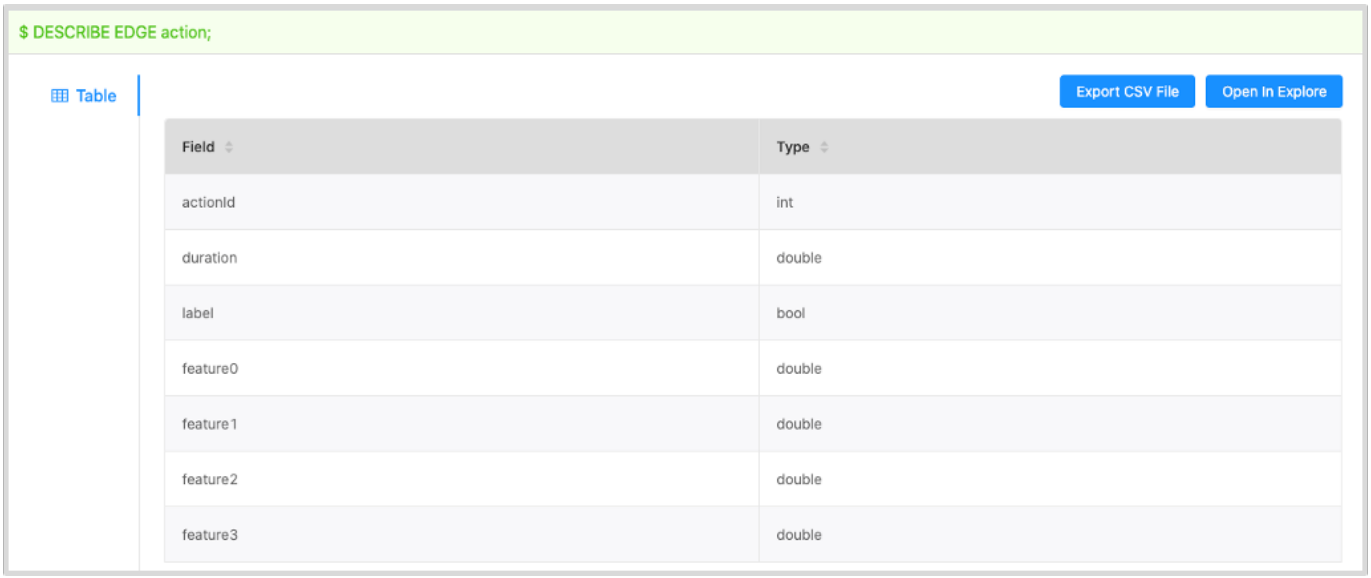
If the preceding statements are executed successfully, the schema is created. You can run the statements as follows to view the schema.

```
 -- To list all the tags in the current graph space
nebula> SHOW TAGS;

-- To list all the edge types in the current graph space
nebula> SHOW EDGES;

-- To view the definition of the tags and edge types
nebula> DESCRIBE TAG user;
nebula> DESCRIBE TAG course;
nebula> DESCRIBE EDGE action;
```

If the schema is created successfully, in the result window, you can see the definition of the tags and edge types. For example, this figure shows the result of the `DESCRIBE EDGE action` statement.



Next to do

When a schema is created, you can import data.

Last update: April 15, 2021

5.3.4 Import data

After CSV files of data and a schema are created, you can use the **Import** page to bulk import vertex and edge data into Nebula Graph for graph exploration and data analysis.

Prerequisites

To bulk import data, do a check of these:

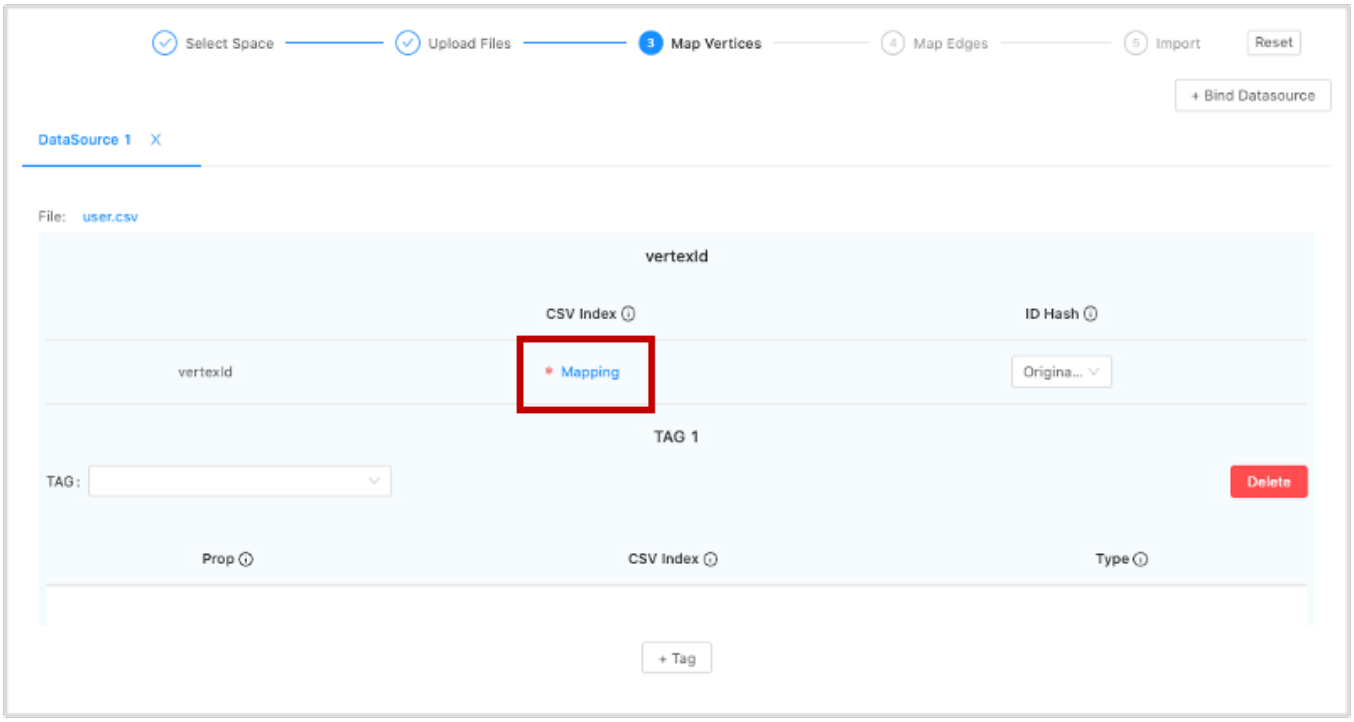
- Studio is connected to Nebula Graph.
- A schema is created.
- CSV files for vertex and edge data separately are created.
- Your account has privileges of GOD, ADMIN, DBA, or USER.

Procedure

To bulk import data, follow these steps:

1. In the toolbar, click the **Import** tab.
2. On the **Select Space** page, choose a graph space name. In this example, **mooc_actions** is used. And then click the **Next** button.
3. On the **Upload Files** page, click the **Upload Files** button and then choose CSV files. In this example, `user.csv`, `course.csv`, and `actions.csv` are chosen.

   NOTE: You can choose multiple CSV files at the same time.
4. On the **Select Files** page, do a check of the file size and click **Preview** or **Delete** in the **Operations** column to make sure that all source data is correct. And then click the **Next** button.
5. On the **Map Vertices** page, click the **+ Bind Datasource** button, and in the dialog box, choose a CSV file. In this example, `user.csv` or `course.csv` is chosen.
6. In the **DataSource X** tab, click the **+ Tag** button.
7. In the **vertexId** section, do these operations: a. In the **CSV Index** column, click **Mapping**.



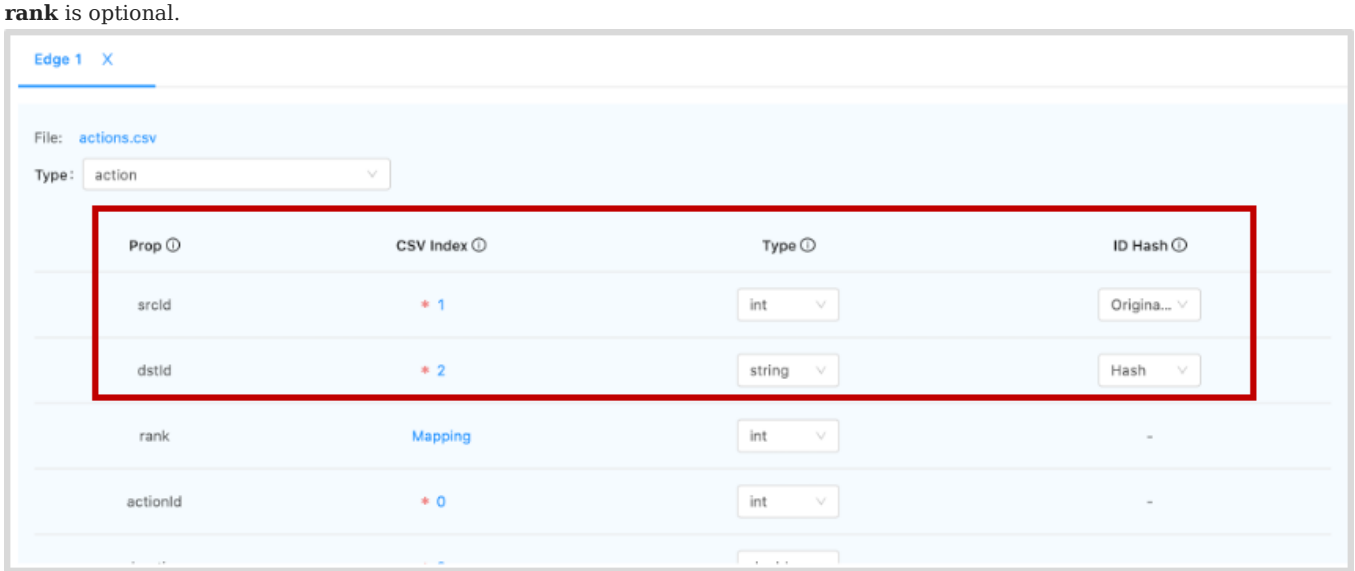
b. In the dialog box, choose a column from the CSV file. In this example, the only one cloumn of `user.csv` is chosen to generate VIDs representing users and the `courseName` column of `course.csv` is chosen to generate VIDs representing courses.

NOTE: VIDs are unique in one graph space. For more information about VIDs, see Vertex Identifier and Partition.

c. In the **ID Hash** column, choose how to generate VIDs. If the source data is of the `int` type, choose **Original ID**. If the source data is of the `string`, `double`, or `bool` type, choose **Hash**.

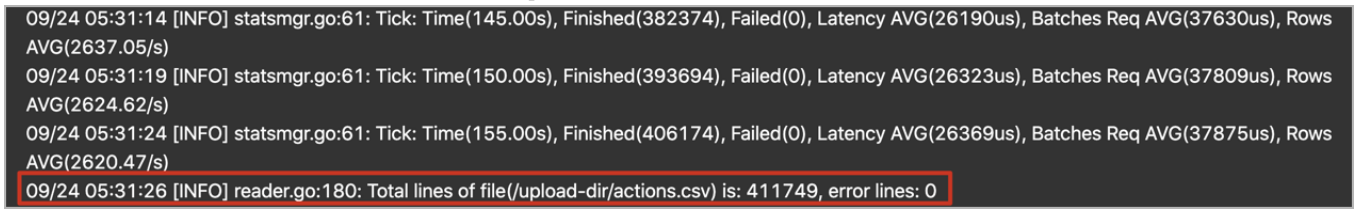
8. In the **TAG 1** section, do these operations:
   a. In the **TAG** drop-down list, choose a tag name. In this example, **user** is used for the `user.csv` file, and **course** is used for the `course.csv` file.
   b. In the property list, click **Mapping** to choose a data column from the CSV file as the value of a property. In this example, the only column of the `user.csv` file is chosen to be the value of the `userId` property of the `user` tag. For the **course** tag, choose **Column 0** for the `courseId` property and set its type to **int**, then choose **Column 1** for the `courseName` property and set its type to **string**.

9. (Optional) If necessary, repeat Step 5 through Step 8 for more tags.
10. When the configuration is done, click the **Next** button.
    When **Config validation was successful** prompts, data mapping for the vertices is successful.
11. On the **Map Edges** page, click the **+ Bind Datasource** button, and in the dialog box, choose a CSV file. In this example, the `actions.csv` file is chosen.
12. In the **Type** drop-down list, choose an edge type name. In this example, **action** is chosen.
13. In the property list, click **Mapping** to choose a column from the `actions.csv` file as values of a property for the edges. **srcId** and **dstId** are the VIDs of the source vertex and destination vertex of an edge. The VID processing must be the same as that of the tag settings. In this example, **srcId** must be set to the VIDs of the users and **dstId** must be set to the VIDs of the courses. **rank** is optional.



14. When the configuration is done, click the **Next** button.
15. On the **Import** page, click the **Start Import** button. On the **log** window, you can see the import progress. The consumed time is related to the data volume. During data import, you can click the **Stop Import** button to stop data import. When the **log**

window shows information as follows, the data import is done.



Next to do

When the data are imported to Nebula Graph, you can query graph data.

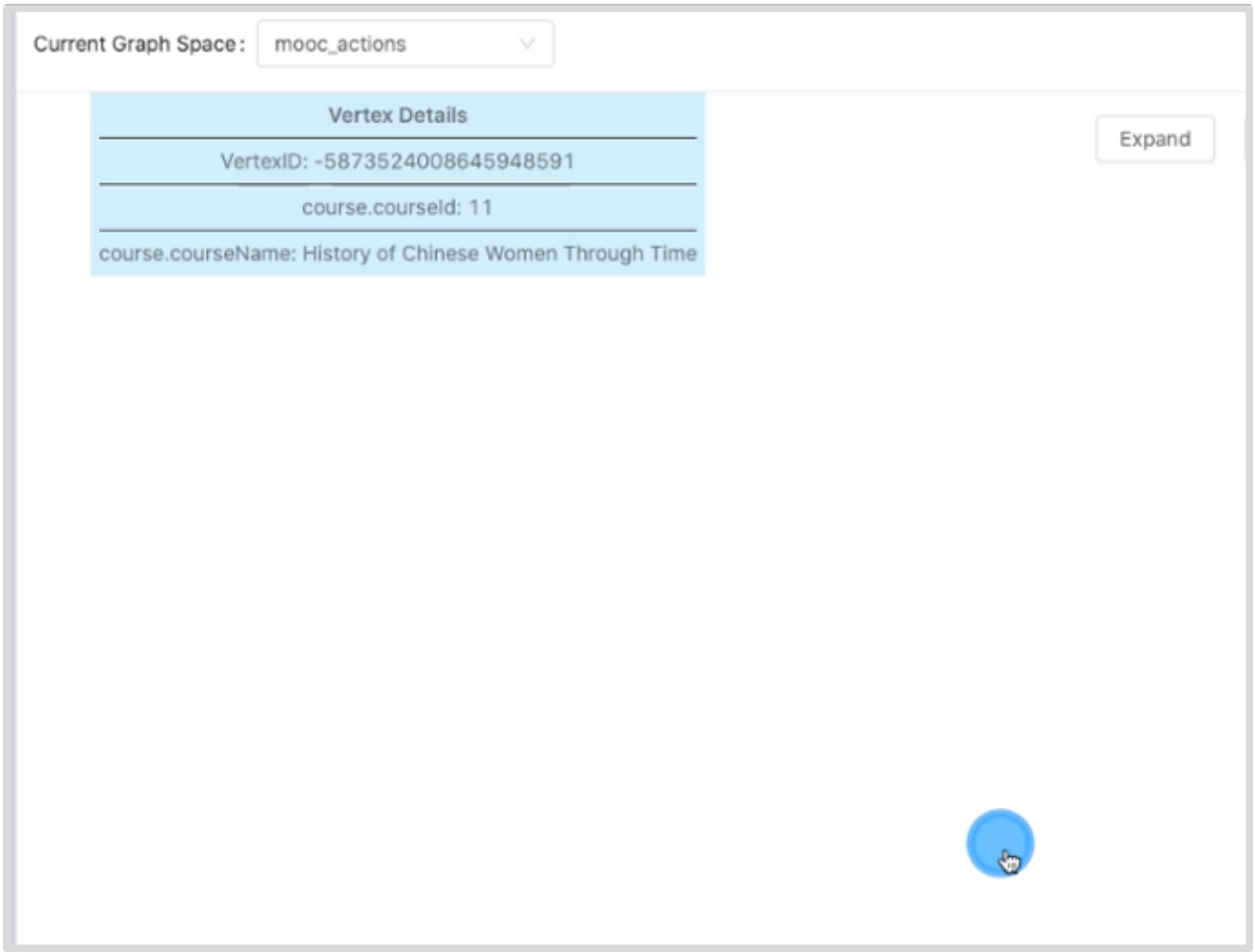
Last update: April 8, 2021

5.3.5 Query graph data

When data is imported, you can use **Console** or **Explore** to query graph data.

For example, if you want to query the properties of the course named *History of Chinese Women Through Time*, you can perform these optional operations:

- On the **Console** tab: Run `FETCH PROP ON * hash("History of Chinese Women Through Time");`. The result window shows all the property information of this vertex. When the result returns, click the **Open in Explore** button and then you can view the vertex information in a visualized way.



- On the **Explore** tab: Click the **Start with Vertices** button. In the dialog box, enter **"History of Chinese Women Through Time"**, choose **Hash** to pre-process the VID, and then click the **Add** button. On the board, you can see the vertex. Move your mouse pointer on the vertex to see the vertex details, as shown in the preceding figure.

Last update: April 8, 2021

5.4 Operation guide

5.4.1 Use Schema

Operate graph spaces

When Studio is connected to Nebula Graph, you can create or delete a graph space. You can use the **Console** page or the **Schema** page to do these operations. This article only introduces how to use the **Schema** page to operate graph spaces in Nebula Graph.

STUDIO VERSION

Studio of v1.2.1-beta or later versions supports this function. To update the version, run this command.

```
docker-compose pull && docker-compose up -d
```

PREREQUISITES

To operate a graph space on the **Schema** page of Studio, you must do a check of these:

- The version of Studio is v1.2.1-beta or later.
- Studio is connected to Nebula Graph.
- Your account has the authority of GOD. It means that:
  - If the authentication is enabled in Nebula Graph, you can use `user` and `password` to sign in to Studio.
  - If the authentication is disabled in Nebula Graph, you must use `root` and its password to sign in to Studio.

CREATE A GRAPH SPACE

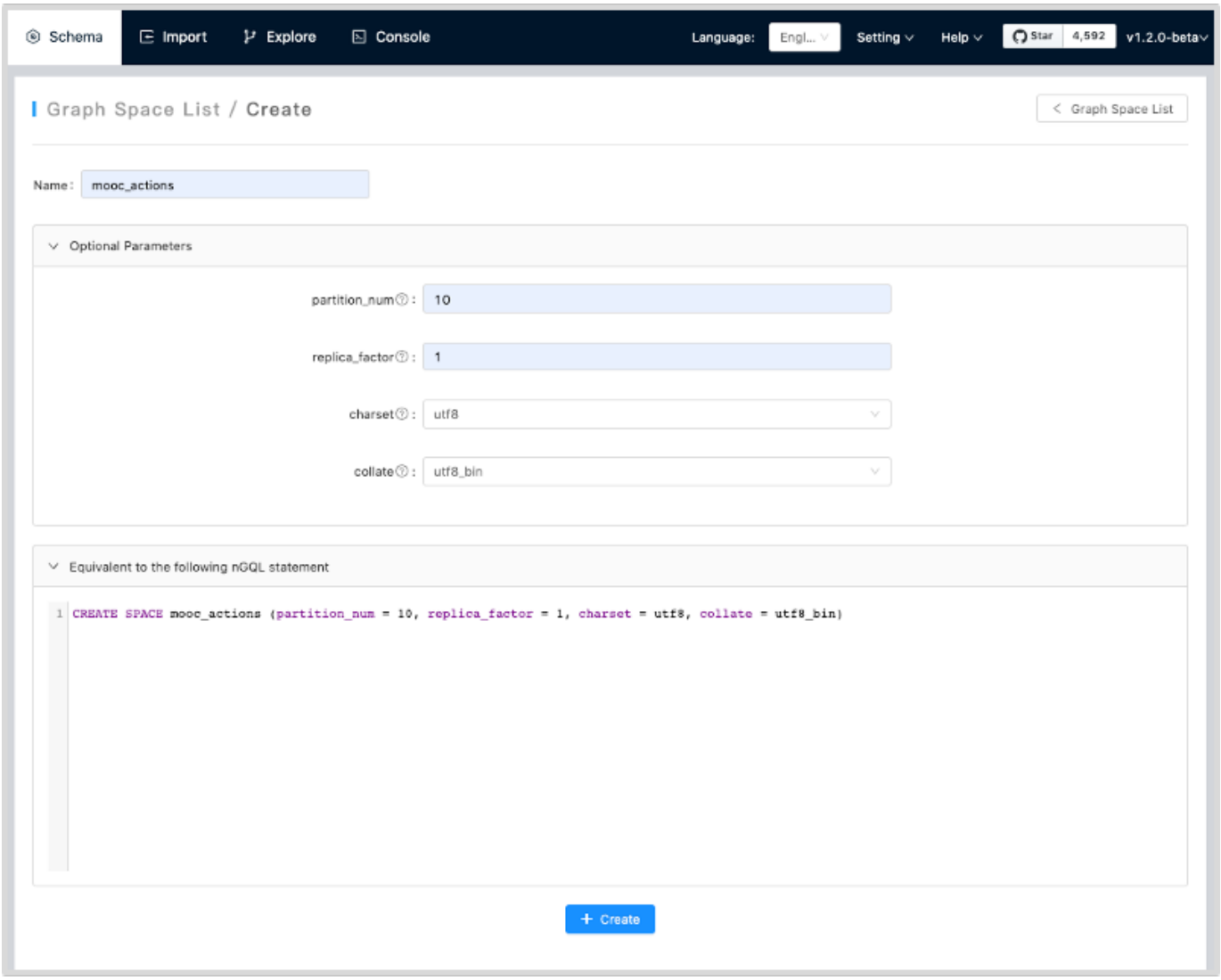
To create a graph space on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. On the **Graph Space List** page, click the **+ Create** button.
3. On the **Create** page, do these settings:

   a. **Name**: Specify a name to the new graph space. In this example, `mooc_actions` is used. The name must be distinct in the database.

   b. **Optional Parameters**: Set `partition_num`, `replica_factor`, `charset`, or `collate`. In this example, these parameters are set to `10`, `1`, `utf8`, and `utf8_bin` separately. For more information, see `CREATE SPACE` syntax.

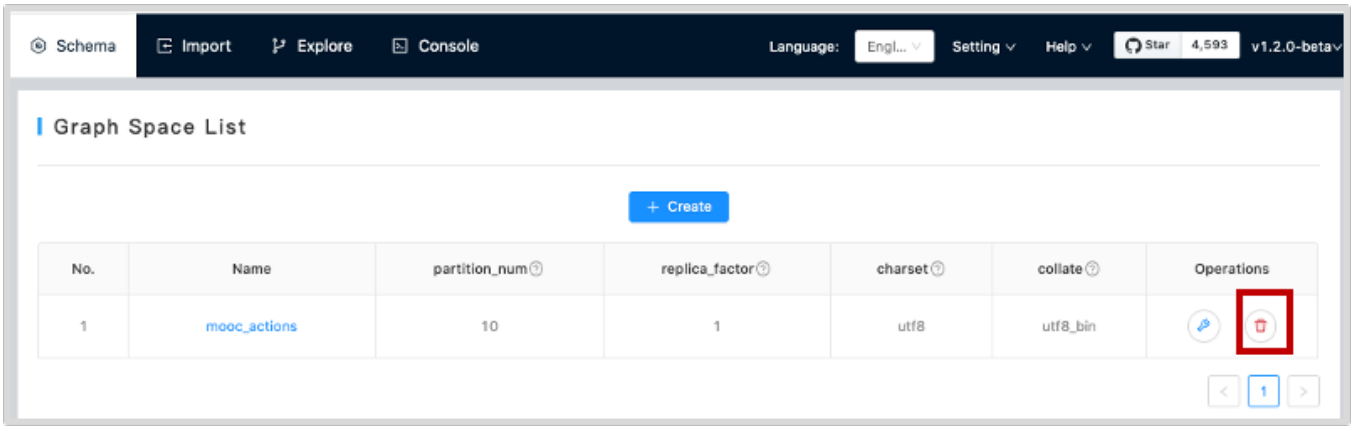
   In the **Equivalent to the following nGQL statement** panel, you can see the statement equivalent to the preceding settings.
4. Confirm the settings and then click the **+ Create** button. If the graph space is created successfully, you can see it on the graph space list.



DELETE A GRAPH SPACE

To delete a graph space on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. In the graph space list, find a graph space and then the button  in the **Operations** column.



3. On the dialog box, confirm the information and then click the **OK** button. When the graph space is deleted successfully, it is removed from the graph space list.

NEXT TO DO

After a graph space is created, you can create or edit a schema, including:

- Operate tags
- Operate edge types
- Operate indexes

Last update: April 15, 2021

Operate tags

After a graph space is created in Nebula Graph, you can create tags. With Studio, you can use the **Console** page or the **Schema** page to create, retrieve, update, or delete tags. This article only introduces how to use the **Schema** page to operate tags in a graph space.

STUDIO VERSION

Studio of v1.2.1-beta or later versions supports this function. To update the version, run this command.

```
docker-compose pull && docker-compose up -d
```

PREREQUISITES

To operate a tag on the **Schema** page of Studio, you must do a check of these:

- The version of Studio is v1.2.1-beta or later.
- Studio is connected to Nebula Graph.
- A graph space is created.
- Your account has the authority of GOD, ADMIN, or DBA.

CREATE A TAG

To create a tag on the **Schema** page, follow these steps:

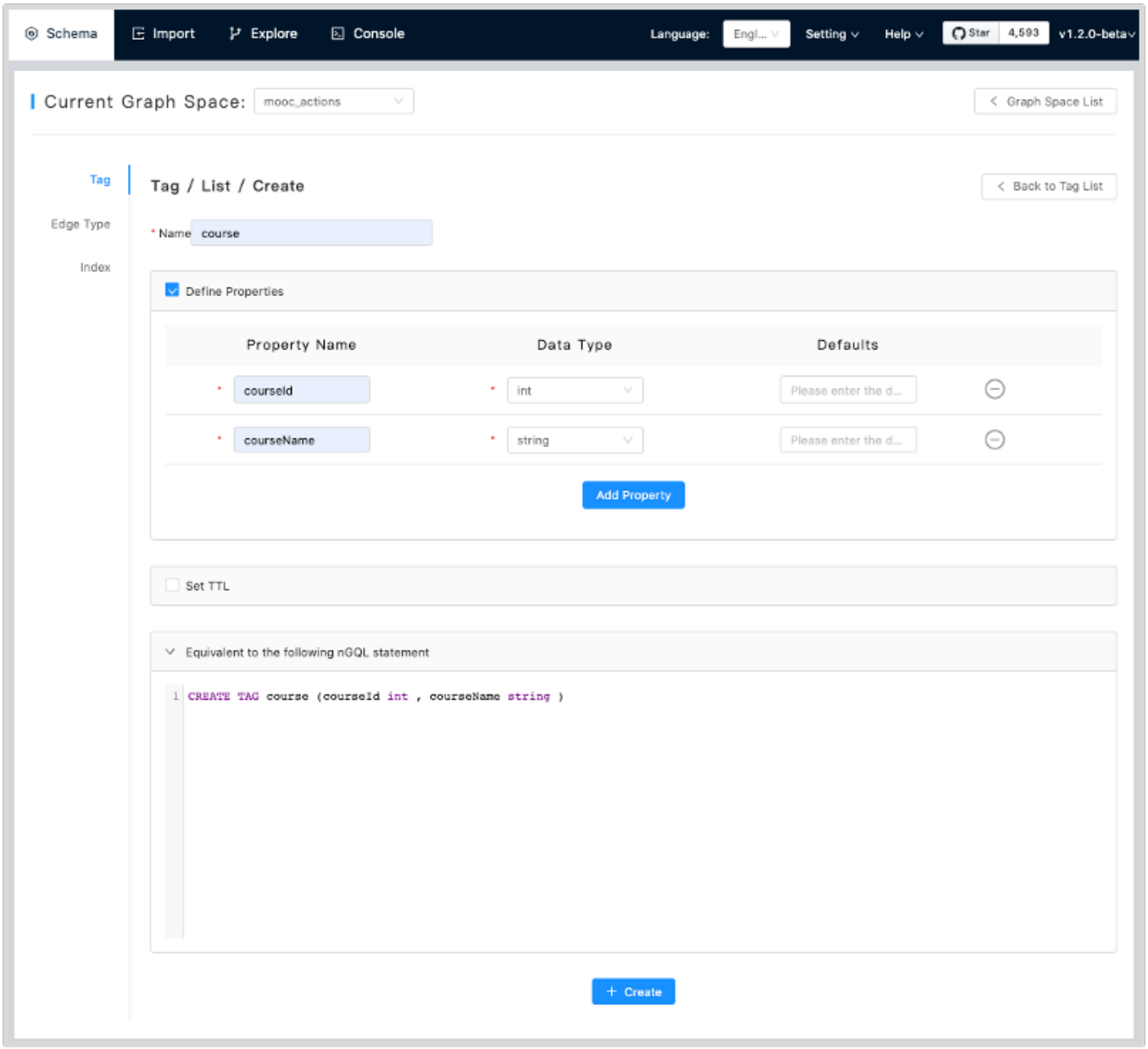
1. In the toolbar, click the **Schema** tab.
2. In the **Graph Space List** page, find a graph space, and then click its name or the button  in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Tag** tab and click the **+ Create** button.
5. On the **Create** page, do these settings:

   a. **Name**: Specify an appropriate name for the tag. In this example, `course` is specified.

   b. (Optional) If necessary, in the upper left corner of the **Define Properties** panel, click the check box to expand the panel and do these settings:

   - To define a property: Enter a property name, a data type, and a default value.
   - To add multiple properties: Click the **Add Property** button and define more properties.
   - To cancel a defined property: Besides the **Defaults** column, click the button  .

   c. (Optional) If no index is set for the tag, you can set the TTL configuration: In the upper left corner of the **Set TTL** panel, click the check box to expand the panel and configure `TTL_COL` and `TTL_ DURATION`. For more information about both parameters, see TTL configuration.
6. When the preceding settings are completed, in the **Equivalent to the following nGQL statement** panel, you can see the nGQL statement equivalent to these settings.



7. Confirm the settings and then click the **+ Create** button.

When the tag is created successfully, the **Define Properties** panel shows all its properties on the list.

EDIT A TAG

To edit a tag on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. In the **Graph Space List** page, find a graph space, and then click its name or the button  in the **Operations** column.
3. In **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Tag** tab, find a tag and then the button  in the **Operations** column.
5. On the **Edit** page, do these settings:
   - To edit a property: On the **Define Properties** panel, find a property, click **Edit**, and then change the data type or the default value.
   - To delete a property: On the **Define Properties** panel, find a property and then click **Delete**.
   - To add more properties: On the **Define Properties** panel, click the **Add Property** button to add a new property.
   - To set the TTL configuration: In the upper left corner of the **Set TTL** panel, click the check box and then set the TTL configuration.
   - To edit the TTL configuration: On the **Set TTL** panel, click **Edit** and then change the configuration of `TTL_COL` and `TTL_DURATION`.
   - To delete the TTL configuration: When the **Set TTL** panel is expanded, in the upper left corner of the panel, click the check box to delete the configuration.
6. When the configuration is done, in the **Equivalent to the following nGQL statement** panel, you can see the equivalent `ALTER TAG` statement.

DELETE A TAG

To delete a tag on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. In **Graph Space List**, find a graph space, and then click its name or the button  in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Tag** tab, find a tag and then the button  in the **Operations** column.

NEXT TO DO

After the tag is created, you can use the **Console** page to insert vertex data one by one manually or use the **Import** page to bulk import vertex data.

Last update: April 15, 2021

Operate edge types

After a graph space is created in Nebula Graph, you can create edge types. With Studio, you can choose to use the **Console** page or the **Schema** page to create, retrieve, update, or delete edge types. This article only introduces how to use the **Schema** page to operate edge types in a graph space.

STUDIO VERSION

Studio of v1.2.1-beta or later versions supports this function. To update the version, run this command.

```
docker-compose pull && docker-compose up -d
```

PREREQUISITES

To operate an edge type on the **Schema** page of Studio, you must do a check of these:

- The version of Studio is v1.2.1-beta or later.
- Studio is connected to Nebula Graph.
- A graph space is created.
- Your account has the authority of GOD, ADMIN, or DBA.

CREATE AN EDGE TYPE

To create an edge type on the **Schema** page, follow these steps:

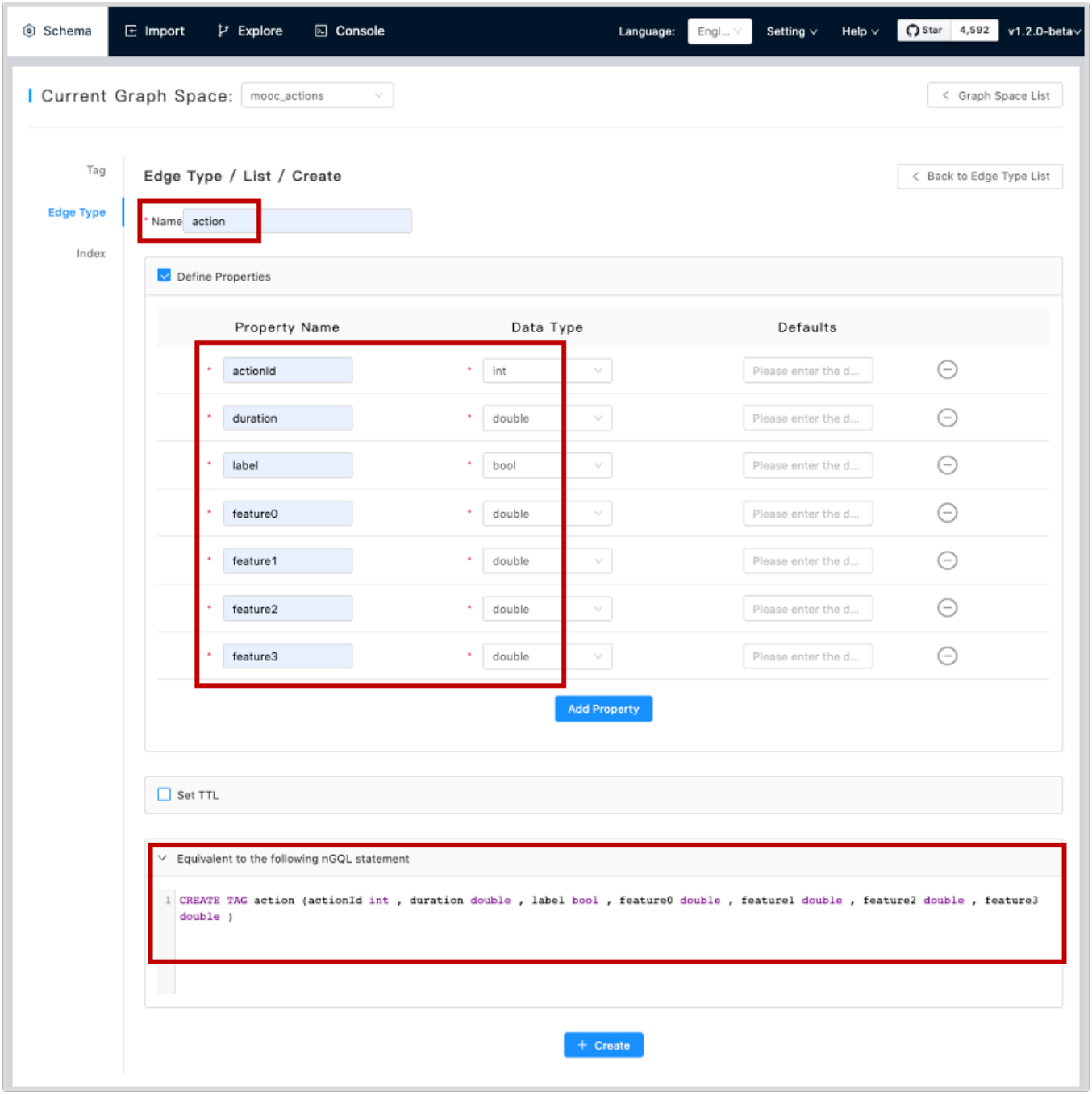
1. In the toolbar, click the **Schema** tab.
2. In the **Graph Space List** page, find a graph space and then click its name or click the button  in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Edge Type** tab and click the **+ Create** button.
5. On the **Create** page, do these settings:

   a. **Name**: Specify an appropriate name for the edge type. In this example, `action` is used.

   b. (Optional) If necessary, in the upper left corner of the **Define Properties** panel, click the check box to expand the panel and do these settings:

   - To define a property: Enter a property name, a data type, and a default value.
   - To add multiple properties: Click the **Add Property** button and define more properties.
   - To cancel a defined property: Besides the **Defaults** column, click the button  .

   c. (Optional) If no index is set for the edge type, you can set the TTL configuration: In the upper left corner of the **Set TTL** panel, click the check box to expand the panel, and configure `TTL_COL` and `TTL_ DURATION`. For more information about both parameters, see TTL configuration.
6. When the preceding settings are completed, in the **Equivalent to the following nGQL statement** panel, you can see the nGQL statement equivalent to these settings.



7. Confirm the settings and then click the **+ Create** button.

When the edge type is created successfully, the **Define Properties** panel shows all its properties on the list.

EDIT AN EDGE TYPE

To edit an edge type on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. In the **Graph Space List** page, find a graph space and then click its name or click the button  in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Edge Type** tab, find an edge type and then click the button  in the **Operations** column.
5. On the **Edit** page, do these operations:
   - To edit a property: On the **Define Properties** panel, find a property, click **Edit**, and then change the data type or the default value.
   - To delete a property: On the **Define Properties** panel, find a property, click **Delete**.
   - To add more properties: On the **Define Properties** panel, click the **Add Property** button to add a new property.
   - To set the TTL configuration: In the upper left corner of the **Set TTL** panel, click the check box and then set TTL.
   - To edit the TTL configuration: On the **Set TTL** panel, click **Edit** and then change the configuration of `TTL_COL` and `TTL_DURATION`.
   - To delete the TTL configuration: When the **Set TTL** panel is expanded, in the upper left corner of the panel, click the check box to delete the configuration.
6. When the configuration is done, in the **Equivalent to the following nGQL statement** panel, you can see the equivalent `ALTER EDGE` statement.

DELETE AN EDGE TYPE

To delete an edge type on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. In **Graph Space List**, find a graph space and then click its name or click the button  in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Edge Type** tab, find an edge type and then click the button  in the **Operations** column.

NEXT TO DO

After the edge type is created, you can use the **Console** page to insert edge data one by one manually or use the **Import** page to bulk import edge data.

Last update: April 15, 2021

Operate Indexes

You can create an index for a tag and/or an edge type. An index lets traversal start from vertices or edges with the same property and it can make a query more efficient. You can create two index types: Tag Index and Edge Type Index. With Studio, you can use the **Console** page or the **Schema** page to create, retrieve, and delete indexes. This article introduces how to use the **Schema** page to operate an index.

> **NOTE**: You can create an index when a tag or an edge type is created. But an index can decrease the write speed during data import. We recommend that you import data firstly and then create and rebuild an index. For more information, see nGQL Manual.

STUDIO VERSION

Studio of v1.2.1-beta or later versions supports this function. To update the version, run this command.

```
docker-compose pull && docker-compose up -d
```

PREREQUISITES

To operate an index on the **Schema** page of Studio, you must do a check of these:

- The version of Studio is v1.2.1-beta or later.
- Studio is connected to Nebula Graph.
- A graph space, tags, and edge types are created.
- Your account has the authority of GOD, ADMIN, or DBA.

CREATE AN INDEX

To create an index on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. On the **Graph Space List** page, find a graph space, and then click its name or the button in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Index** tab and then click the **+ Create** button.
5. On the **Create** page, do these settings:

   a. **Index Type**: Choose to create an index for a tag or for an edge type. In this example, **Edge Type** is chosen.

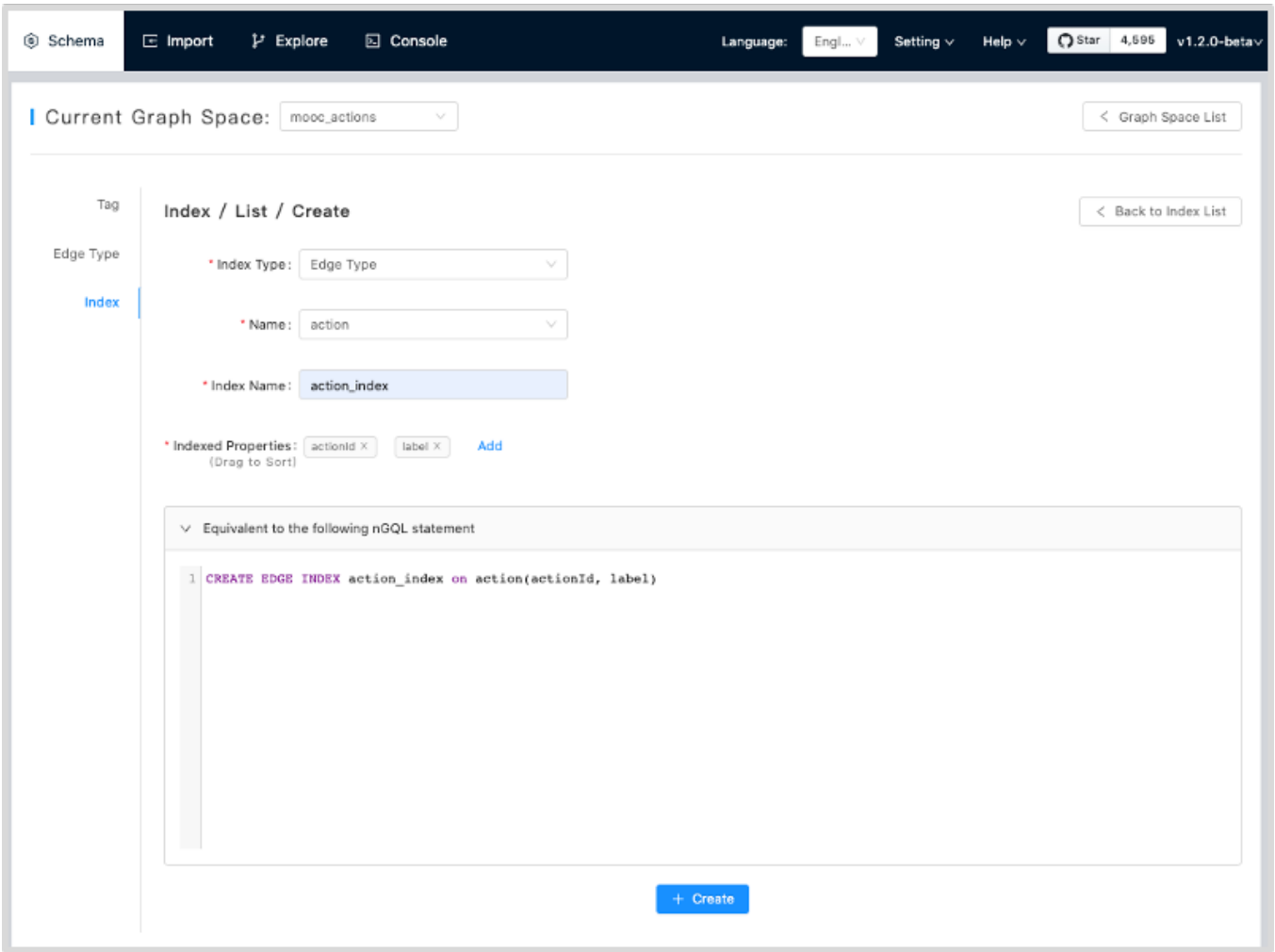
   b. **Name**: Choose a tag name or an edge type name. In this example, **action** is chosen.

   c. **Index Name**: Specify a name for the new index. In this example, **action_index** is used.

   d. **Indexed Properties**: Click **Add**, and then, in the dialog box, choose a property. If necessary, repeat this step to choose more properties. You can drag the properties to sort them. In this example, `actionId` and `label` are chosen.

   > **NOTE**: The order of the indexed properties has an effect on the result of the `LOOKUP` statement. For more information, see nGQL Manual.

   When the settings are done, the **Equivalent to the following nGQL statement** panel shows the statement equivalent to the settings.



6. Confirm the settings and then click the **+ Create** button.
   When an index is created, the index list shows the new index.

VIEW INDEXES

To view indexes on the **Schema** page, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. In the graph space list, find a graph space, and then click its name or the button  in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Index** tab, in the upper left corner, choose an index type, **Tag** or **Edge Type**.
5. In the list, find an index and click its row. All its details are shown in the expanded row.

DELETE AN INDEX

To delete an index on **Schema**, follow these steps:

1. In the toolbar, click the **Schema** tab.
2. In the graph space list, find a graph space, and then click its name or the button  in the **Operations** column.
3. In the **Current Graph Space** field, confirm the name of the graph space. If necessary, you can choose another name to change the graph space.
4. Click the **Index** tab, find an index and then the button  in the **Operations** column.

Last update: April 15, 2021

5.4.2 Use Console

Open in Explore

With the **Open in Explore** function, you can run nGQL statements on the **Console** page to query vertex or edge data and then view the result on the **Explore** page in a visualized way.

STUDIO VERSION

Studio of v1.2.1-beta or later versions supports this function. To update the version, run this command.

```
docker-compose pull && docker-compose up -d
```

PREREQUISITES

To use the **Open in Explore** function, you must do a check of these:

- The version of Studio is v1.2.1-beta or later.
- Studio is connected to Nebula Graph.
- A dataset exists in the database.

QUERY AND VISUALIZE EDGE DATA

To query edge data on the **Console** page and then view the result on the **Explore** page, follow these steps:

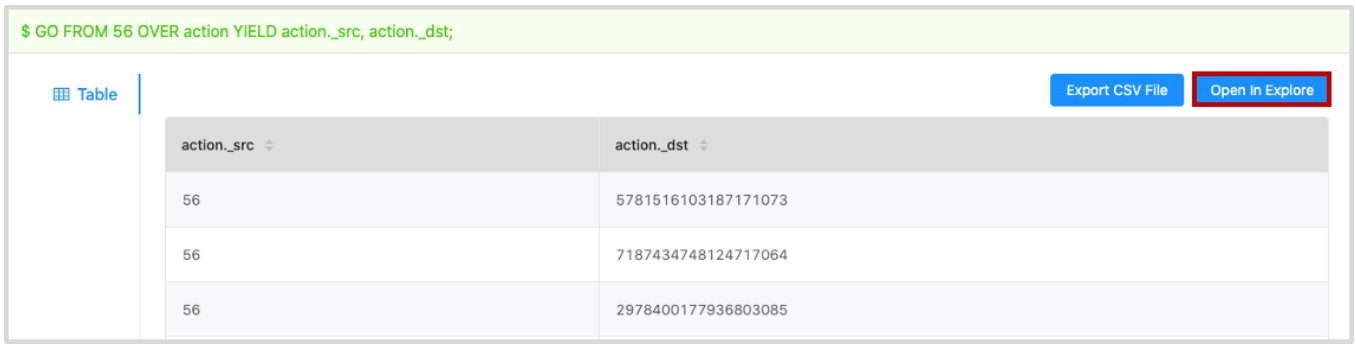
1. In the toolbar, click the **Console** tab.
2. In the **Current Graph Space** field, choose a graph space name. In this example, **mooc_actions** is chosen.
3. In the input box, enter an nGQL statement and click the button  .

   NOTE: The queried result must contain the VIDs of the source vertex and the destination vertex of an edge.

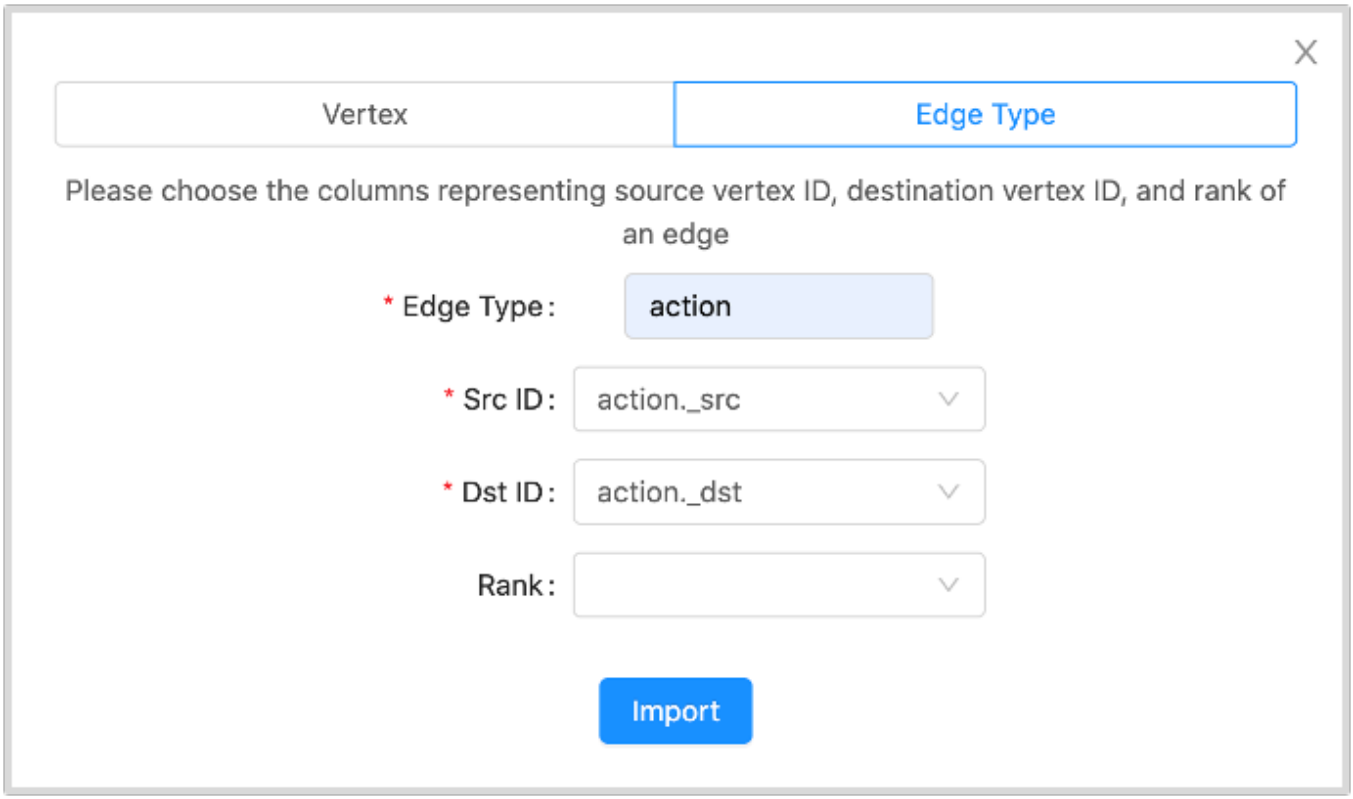
   Here is an nGQL statement example.

   ```
   nebula> GO FROM 56 OVER action YIELD action._src, action._dst;
   ```

   The queried result gives the edges between the user with ID 56 and the courses that he/she takes on the MOOC platform, as shown in this figure.

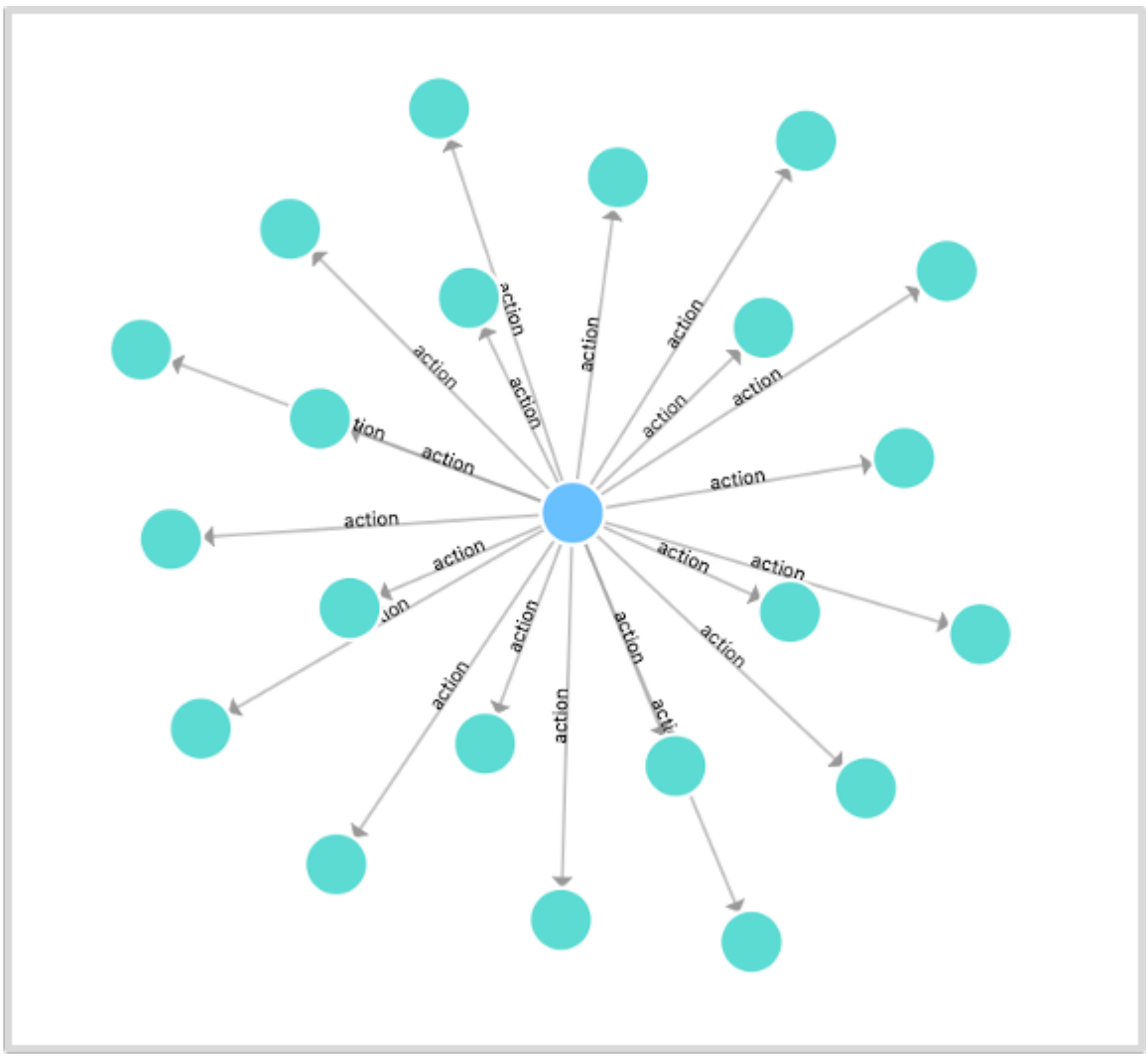


4. Click the **Open in Explore** button.
5. In the dialog box, configure as follows:
   a. Click **Edge Type**.
   b. In the **Edge Type** field, enter an edge type name. In this example, `action` is used.
   c. In the **Src ID** field, choose a column name from the result table representing the VIDs of the source vertices. In this example, `action._src` is chosen.
   d. In the **Dst ID** field, choose a column name from the result table representing the VIDs of the destination vertices. In this example, `action._dst` is chosen.
   e. (Optional) If the result table contains the ranking information of the edges, in the **Rank** field, choose a column name representing the `rank` of the edges. If no ranking information exists in the result, leave the **Rank** field blank.
   f. When the configuration is done, click the **Import** button.



6. (Optional) If some data exists on the board of **Explore**, choose a method to insert data:
   - **Incremental Insertion**: Click this button to add the result to the existing data on the board.
   - **Insert After Clear**: Click this button to clear the existing data from the board and then add the data to the board.

When the data is inserted, you can view the visualized representation of the edge data.



QUERY AND VISUALIZE VERTEX DATA

To query vertex data on the **Console** page and then view the result on the **Explore** page, follow these steps:

1. In the toolbar, click the **Console** tab.
2. In the **Current Graph Space** field, choose a graph space name. In this example, **mooc_actions** is chosen.
3. In the input box, enter an nGQL statement and click the button  .

   NOTE: The queried result must contain the VIDs of the vertices.

   Here is an nGQL statement example.

   ```
   nebula> FETCH PROP ON * hash("Media History and Theory"); -- For the mooc_actions dataset, the VIDs of the course vertices are generated by
   Hash("<courseName>").
   ```

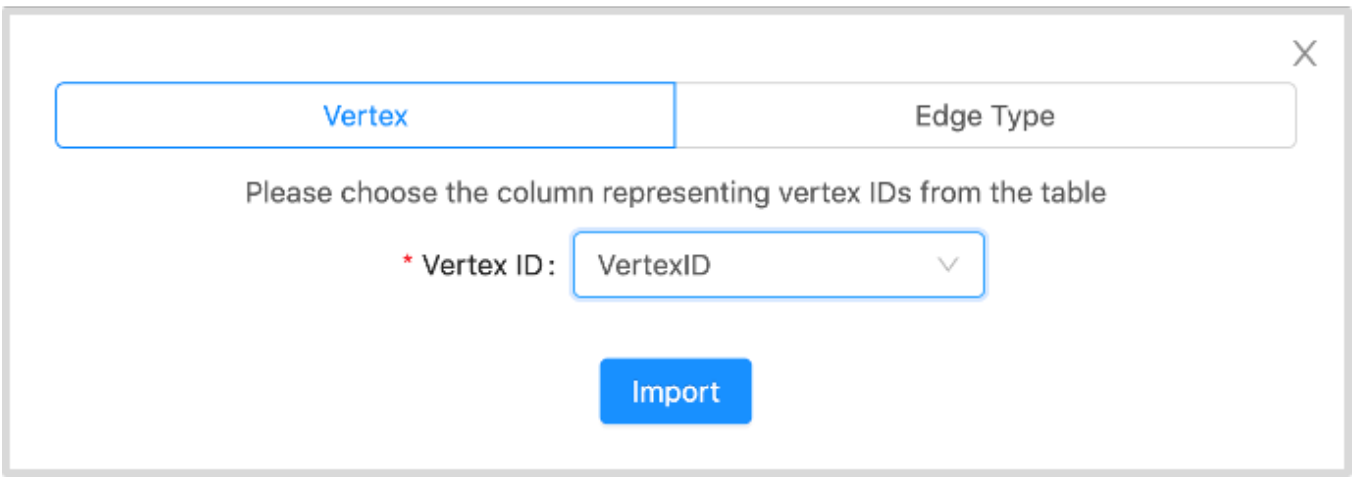
   The queried result gives the course with ID 8, as shown in this figure.

4. Click the **Open in Explore** button.
5. In the dialog box, configure as follows:

   a. Click **Vertex**.

   b. In the **Vertex ID** field, choose a column name from the result table representing the VIDs of the vertices. In this example, `VertexID` is chosen.

   c. When the configuration is done, click the **Import** button.



6. (Optional) If some data exists on the board of **Explore**, choose a method to insert data:
   - **Incremental Insertion**: Click this button to add the queried result to the existing data on the board.
   - **Insert After Clear**: Click this button to clear the existing data from the board and then add the data.

When the data is inserted, you can view the visualized representation of the vertex data.

NEXT TO DO

On the **Explore** page, you can expand the board to explore and analyze graph data.

Last update: April 8, 2021

View subgraphs

With the **View Subgraphs** function, you can run a `FIND SHORTEST | ALL PATH` statement on the **Console** page to retrieve all the paths or the shortest path between the specified vertices and then view the result on the **Explore** page.

For more information about `FIND SHORTEST | ALL PATH`, see nGQL User Guide.

STUDIO VERSION

Studio of v1.2.1-beta or later versions supports this function. To update the version, run this command.

```
$ docker-compose pull && docker-compose up -d
```

PREREQUISITES

To use the **View Subgraphs** function, you must do a check of these:

- The version of Studio is v1.2.1-beta or later.
- Studio is connected to Nebula Graph.
- A dataset exists in the database. In the example of this article, the **mooc_actions** dataset is used. For more information, see Import data.

PROCEDURE

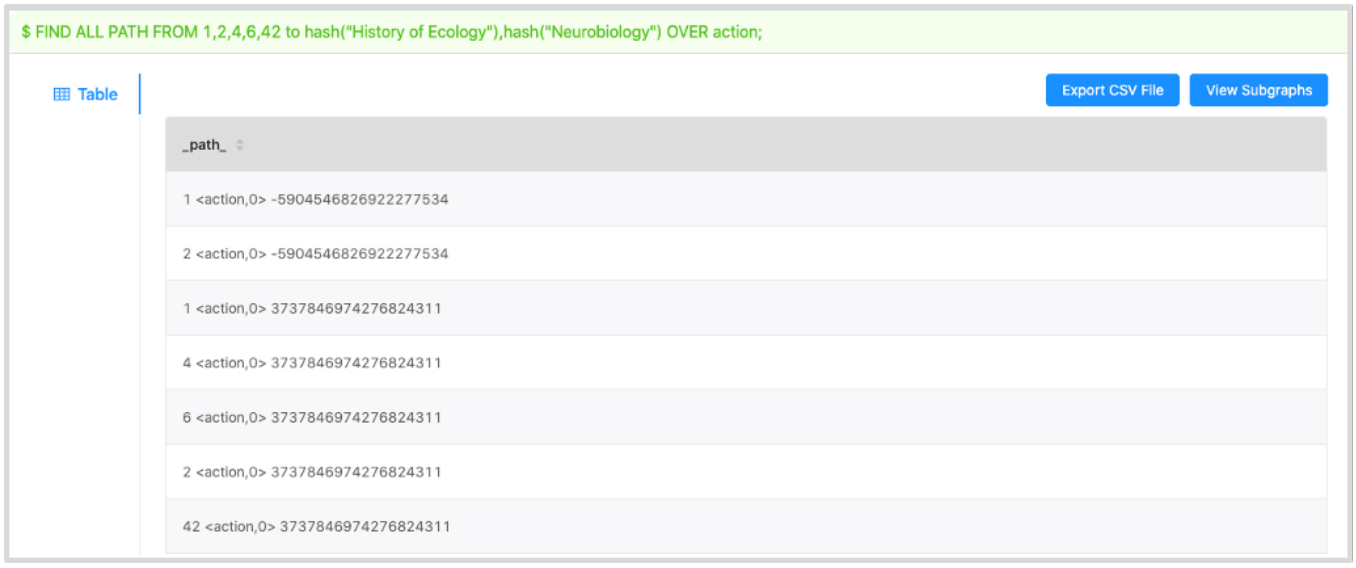
To query the paths on the **Console** page and then view them on the **Explore** page, follow these steps:

1. In the navigation bar, click the **Console** tab.
2. In the **Current Graph Space** dropdown list, choose a graph space name. In this example, **mooc_actions** is chosen.
3. In the input box, enter a `FIND SHORTEST PATH` or `FIND ALL PATH` statement and click **Run**.

   Here is an nGQL statement example.

   ```
   nebula> FIND ALL PATH FROM 1,2,4,6,42 to hash("History of Ecology"),hash("Neurobiology") OVER action; -- For the mooc_actions dataset, the VIDs of the
   course vertices are generated by the hash("<courseName>") function.
   ```

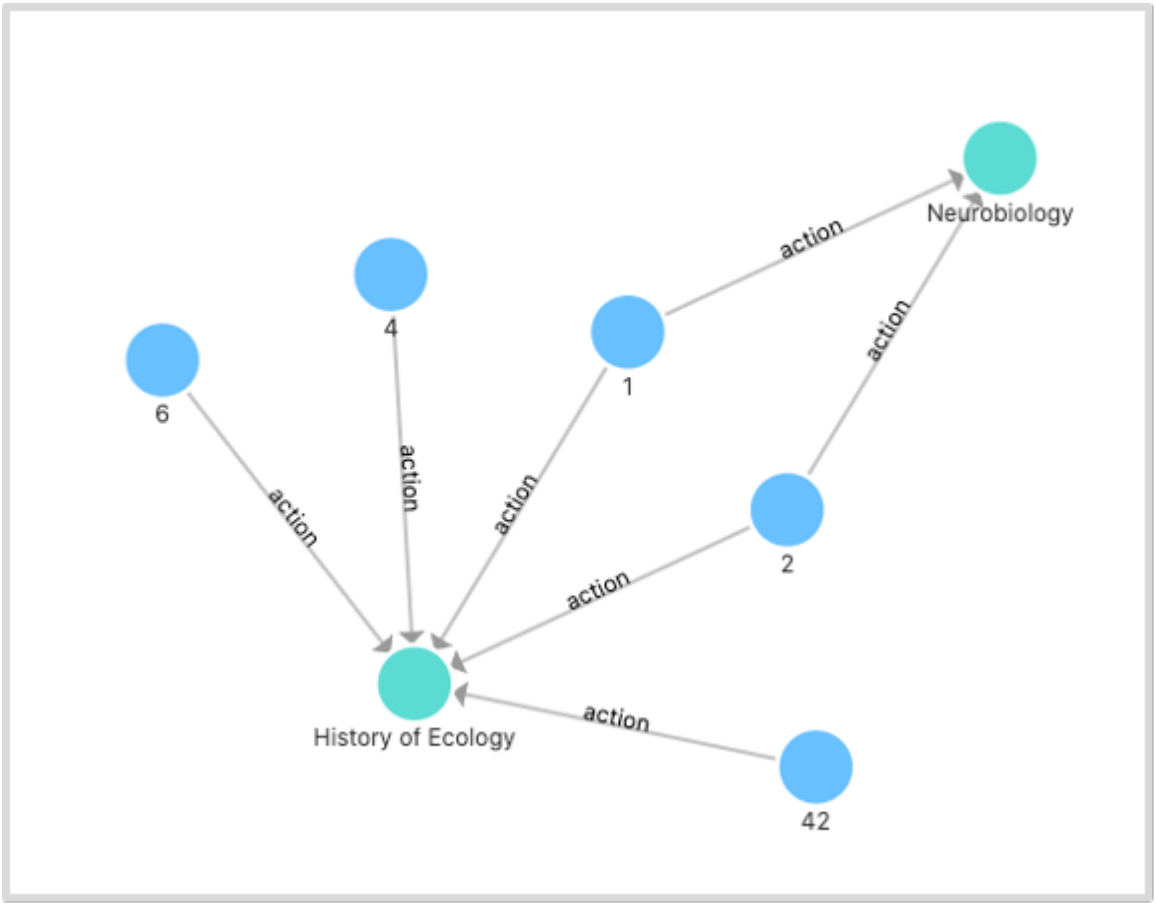
   The queried result gives all the paths from the specified user vertices to the course vertices, as shown in this figure.



4. Click the **View Subgraphs** button.
5. (Optional) If some data exists on the board of **Explore**, choose a method to insert data:

   - **Incremental Insertion**: Click this button to add the result to the existing data on the board. - **Insert After Clear**: Click this button to clear the existing data from the board and then add the data to the board.

When the data is inserted, you can view the visualized representation of the paths.



NEXT TO DO

On the **Explore** page, you can expand the graph to explore and analyze graph data.

Last update: April 8, 2021

6. Contributions

6.1 Contribute to Documentation

Contributing to the **Nebula Graph** documentation can be a rewarding experience. We welcome your participation to help make the documentation better!

6.1.1 How to Contribute to the Docs

There are many ways to contribute:

- Raise a documentation issue on GitHub.
- Fork the documentation, make changes or add new content on your local branch, and submit a pull request (PR) to the master branch for the docs.

Last update: April 8, 2021

6.2 Cpp Coding Style

Please Refer to Google C++ Style Guide.

Last update: April 8, 2021

6.3 How to Contribute

This topic uses the `vesoft-inc/nebula` repository as an example to introduce how to contribute to Nebula Graph related projects and their documentation. For the complete project list, visit the vesoft-inc repositories on GitHub.

6.3.1 Sign the CLA

Click the **Sign in with Github to agree** button to sign the CLA.

What is CLA?

6.3.2 Step 1: Fork in the Cloud

1. Visit https://github.com/vesoft-inc/nebula.
2. Click `Fork` button (top right) to establish a cloud-based fork.

6.3.3 Step 2: Clone Fork to Local Storage

Define a local working directory:

```
# Define your working directory
working_dir=$HOME/Workspace
```

Set `user` to match your Github profile name:

```
user={your Github profile name}
```

Create your clone:

```
mkdir -p $working_dir
cd $working_dir
git clone https://github.com/$user/nebula.git
# the following is recommended
# or: git clone git@github.com:$user/nebula.git

cd $working_dir/nebula
git remote add upstream https://github.com/vesoft-inc/nebula.git
# or: git remote add upstream git@github.com:vesoft-inc/nebula.git

# Never push to upstream master since you do not have write access.
git remote set-url --push upstream no_push

# Confirm that your remotes make sense:
# It should look like:
# origin    git@github.com:$(user)/nebula.git (fetch)
# origin    git@github.com:$(user)/nebula.git (push)
# upstream  https://github.com/vesoft-inc/nebula (fetch)
# upstream  no_push (push)
git remote -v
```

6.3.4 Step 3: Create a Branch

Get your local master up to date:

```
cd $working_dir/nebula
git fetch upstream
git checkout master
git rebase upstream/master
```

Checkout a new branch from master:

```
git checkout -b myfeature
```

NOTE: Because your PR often consists of several commits, which might be squashed while being merged into upstream, we strongly suggest you open a separate topic branch to make your changes on. After merged, this topic branch could be just abandoned, thus you could synchronize your master branch with upstream easily with a rebase like above. Otherwise, if you commit your changes directly into master, maybe you must use a hard reset on the master branch, like:

```
git fetch upstream
git checkout master
git reset --hard upstream/master
git push --force origin master
```

6.3.5 Step 4: Develop

Code Style

We adopt `cpplint` to make sure that the project conforms to Google's coding style guides. The checker will be implemented before the code is committed.

Unit Tests Required

Please add unit tests for your new features or bugfixes.

Build Your Code with Unit Tests Enable

Please refer to the build source code documentation to compile.

> Make sure you have enabled the build of unit tests by setting `-DENABLE_TESTING=ON`.

Run Tests

In the root folder of `nebula` , run the following command:

```
ctest -j$(nproc)
```

6.3.6 Step 5: Bring Your Branch Update to Date

```
# While on your myfeature branch.
git fetch upstream
git rebase upstream/master
```

You need to bring the head branch up to date after other collaborators merge pull requests to the base branch.

6.3.7 Step 6: Commit

Commit your changes.

```
git commit
```

Likely you'll go back and edit/build/test some more than `commit --amend` in a few cycles.

6.3.8 Step 7: Push

When ready to review (or just to establish an offsite backup or your work), push your branch to your fork on `github.com`:

```
git push origin myfeature
```

6.3.9 Step 8: Create a Pull Request

1. Visit your fork at https://github.com/$user/nebula (replace `$user` obviously).
2. Click the `Compare & pull request` button next to your `myfeature` branch.

6.3.10 Step 9: Get a Code Review

Once your pull request has been opened, it will be assigned to at least two reviewers. Those reviewers will do a thorough code review to make sure that the changes meet the repository's contributing guidelines and other quality standards.

Last update: April 8, 2021

6.4 Pull Request and Commit Message Guidelines

This document describes the commit message and Pull Request style applied to all **Nebula Graph** repositories. Every commit made *directly* to the `master` branch must follow the below guidelines.

6.4.1 Commit Message

```
<type>(<scope>): <subject> // scope is optional, subject is must

                <body> // optional

                <footer> // optional
```

These rules are adopted from the AngularJS commit convention.

- `<Type>` describes the kind of change that this commit is providing.
- `<subject>` is a short description of the change.
- If additional details are required, add a blank line, and then provide explanation and context in paragraph format.

Commit Types

Type	Description
Feature	New features
Fix	Bug fix
Doc	Documentation changes
Style	Formatting, missing semi colons, ...
Refactor	Code cleanup
Test	New tests
Chore	Maintain

6.4.2 Pull Request

When you submit a Pull Request, please include enough details about all changes in the title but keep it concise.

The title of a pull request must briefly describe the changes made.

For very simple changes, you can leave the description blank as there’s no need to describe what will be obvious from looking at the diff. For more complex changes, give an overview of the changes. If the PR fixes an issue, make sure to include the GitHub issue-number in the description.

Pull Request Template

```
What changes were proposed in this pull request?

Why are the changes needed?

Does this PR introduce any user-facing change?

How was this patch tested?
```

Last update: April 8, 2021

7. Appendix

7.1 Comparison Between Cypher and nGQL

7.1.1 Conceptual Comparisons

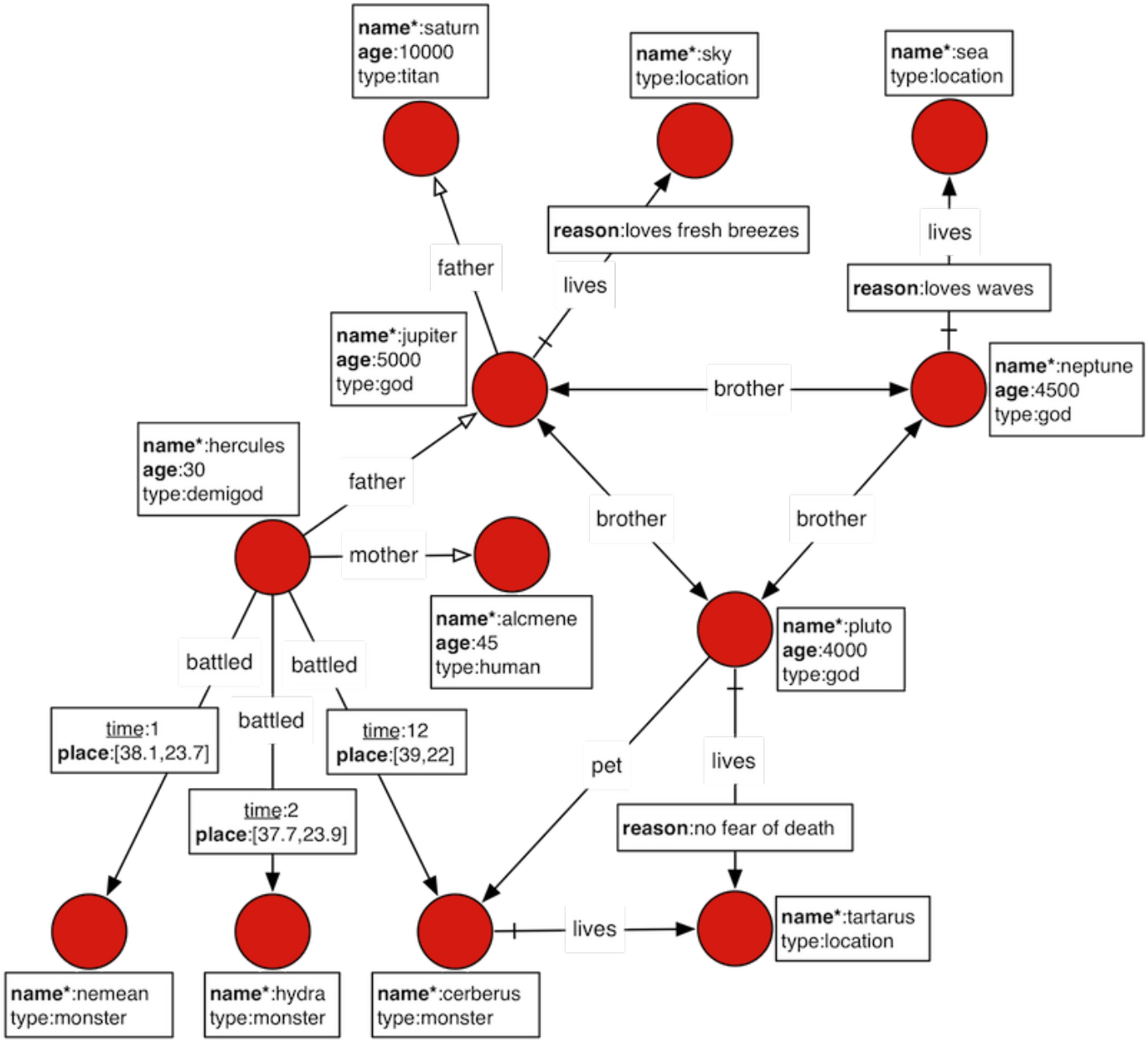
Name	Cypher	nGQL
vertex, node	node	vertex
edge, relationship	relationship	edge
vertex type	label	tag
edge type	relationship type	edge type
vertex identifier	node id generated by default	vid
edge identifier	edge id generated by default	src, dst, rank

7.1.2 Basic Graph Operations

Operations	Cypher	nGQL
List all labels/tags	MATCH (n) RETURN distinct labels(n); call db.labels();	SHOW TAGS
Insert a vertex with a specified type	CREATE (:Person {age: 16})	INSERT VERTEX `<tag_name>` (prop_name_list) VALUES `<vid>` :(prop_value_list)
Insert an edge with specified edge type	CREATE (src)-[rel:LIKES]->(dst) SET rel.prop = V	INSERT EDGE `<edge_type>` (prop_name_list) VALUES `<src_vid>` -> `<dst_vid>` [`@<rank>`]: (prop_value_list)
Delete a vertex	MATCH (n) WHERE ID(n) = vid DETACH DELETE n	DELETE VERTEX `<vid>`
Delete an edge	MATCH ()-[r]->() WHERE ID(r)=edgeID DELETE r	DELETE EDGE `<edge_type>` `<src_vid>` -> `<dst_vid>` [`@<rank>`]
Update a vertex property	SET n.name = V	UPDATE VERTEX `<vid>` SET `<update_columns>`
Fetch vertex prop	MATCH (n) WHERE ID(n) = vid RETURN properties(n)	FETCH PROP ON `<tag_name>` `<vid>`
Fetch edge prop	MATCH (n)-[r]->() WHERE ID(r)=edgeID return properties(r)	FETCH PROP ON `<edge_type>` `<src_vid>` -> `<dst_vid>` [`@<rank>`]
Query a vertex along specified edge type	MATCH (n)-[r:edge_type]->() WHERE ID(n) = vid	GO FROM `<vid>` OVER `<edge_type>`
Query a vertex along specified edge type reversely	MATCH (n)<-[r:edge_type]-() WHERE ID(n) = vid	GO FROM `<vid>` OVER `<edge_type>` REVERSELY
Get the N-Hop along a specified edge type	MATCH (n)-[r:edge_type*N]->() WHERE ID(n) = vid return r	GO N STEPS FROM `<vid>` OVER `<edge_type>`
Find path between two vertices	MATCH p =(a)-[]->(b) WHERE ID(a) = a_vid AND ID(b) = b_vid RETURN p	FIND ALL PATH FROM `<a_vid>` TO `<b_vid>` OVER *

7.1.3 Example Queries

The example queries are based on the graph below:



- Insert data

```
# insert vertex
nebula> INSERT VERTEX character(name, age, type) VALUES hash("saturn"):("saturn", 10000, "titan"), hash("jupiter"):("jupiter", 5000, "god");

# insert edge
nebula> INSERT EDGE father() VALUES hash("jupiter")->hash("saturn"):();

// cypher
cypher> CREATE (src:character {name:"saturn", age: 10000, type:"titan"})
      > CREATE (dst:character {name:"jupiter", age: 5000, type:"god"})
      > CREATE (src)-[rel:father]->(dst)
 ```

- Delete vertex

```ngql
nebula> DELETE VERTEX hash("prometheus");

cypher> MATCH (n:character {name:"prometheus"})
        > DETACH DELETE n
```

- Update vertex

```
nebula> UPDATE VERTEX hash("jesus") SET character.type = 'titan';
```

```
cypher> MATCH (n:character {name:"jesus"})
      > SET n.type = 'titan'
```

- Fetch vertices properties

```
nebula> FETCH PROP ON character hash("saturn");
  ================================================
  | character.name | character.age | character.type |
  ================================================
  | saturn         | 10000         | titan          |
  ------------------------------------------------

cypher> MATCH (n:character {name:"saturn"})
      > RETURN properties(n)
  ╒═══════════════════════════════════════════╕
  │"properties(n)"                            │
  ╞═══════════════════════════════════════════╡
  │{"name":"saturn","type":"titan","age":10000}│
  └───────────────────────────────────────────┘
```

- Find the name of hercules's grandfather

```
nebula> GO 2 STEPS FROM hash("hercules") OVER father YIELD  $$.character.name;
    ====================
    | $$.character.name |
    ====================
    | saturn            |
    --------------------

cypher> MATCH (src:character{name:"hercules"})-[r:father*2]->(dst:character)
      > RETURN dst.name
       ╒═══════════╕
       │"dst.name"│
       ╞═══════════╡
       │"satun"   │
       └───────────┘
```

- Find the name of hercules's father

```
nebula> GO FROM hash("hercules") OVER father YIELD $$.character.name;
    ====================
    | $$.character.name |
    ====================
    | jupiter           |
    --------------------

cypher> MATCH (src:character{name:"hercules"})-[r:father]->(dst:character)
      > RETURN dst.name
       ╒═══════════╕
       │"dst.name"│
       ╞═══════════╡
       │"jupiter" │
       └───────────┘
```

- Find the the centenarians' name.

```
nebula> # coming soon

cypher> MATCH (src:character)
      > WHERE src.age > 100
      > RETURN src.name
       ╒═══════════╕
       │"src.name" │
       ╞═══════════╡
       │  "saturn" │
       ├───────────┤
       │ "jupiter" │
       ├───────────┤
       │ "neptune" │
       ├───────────┤
       │  "pluto"  │
       └───────────┘
```

- Find who live with pluto

```
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | GO FROM $-.place OVER lives REVERSELY WHERE \
      > $$.character.name != "pluto" YIELD $$.character.name AS cohabitants;
    ===============
    | cohabitants |
    ===============
    | cerberus    |
    ---------------

cypher> MATCH (src:character{name:"pluto"})-[r1:lives]->()<-[r2:lives]-(dst:character)
      > RETURN dst.name
```

```
      ╒═══════════╕
      |"dst.name"|
      ╞═══════════╡
      |"cerberus"|
      └───────────┘
```

- Find pluto's brother and their habitations?

```
nebula> GO FROM hash("pluto") OVER brother YIELD brother._dst AS god | \
      > GO FROM $-.god OVER lives YIELD $^.character.name AS Brother, $$.location.name AS Habitations;
    ========================
    | Brother | Habitations |
    ========================
    | jupiter | sky         |
    ------------------------
    | neptune | sea         |
    ------------------------

cypher> MATCH (src:Character{name:"pluto"})-[r1:brother]->(bro:Character)-[r2:lives]->(dst)
      > RETURN bro.name, dst.name
      ╒═══════════════════════════╕
      |"bro.name"    |"dst.name"|
      ╞═══════════════════════════╡
      | "jupiter"    |  "sky"   |
      ├──────────────┼──────────┤
      | "neptune"    | "sea"    |
      └──────────────┴──────────┘
```

Last update: April 8, 2021

7.2 Comparison Between Gremlin and nGQL

7.2.1 Introduction to Gremlin

Gremlin is a graph traversal language developed by Apache TinkerPop. It can be either declarative or imperative. Gremlin is Groovy-based, but has many language variants that allow developers to write Gremlin queries natively in many modern programming languages such as Java, JavaScript, Python, Scala, Clojure and Groovy.

7.2.2 Introduction to nGQL

Nebula Graph introduces its own query language, nGQL, which is a declarative, textual query language like SQL, but for graphs. Unlike SQL, `nGQL` is all about expressing graph patterns. The features of `nGQL` are as follows:

- Syntax is close to SQL, but not exactly the same (Easy to learn)
- Expandable
- Keyword is case insensitive
- Support basic graph traverse
- Support pattern matching
- Support aggregation
- Support graph mutation
- Support distributed transaction (future release)
- Statement composition, but **NO** statement embedding (Easy to read)

7.2.3 Conceptual Comparisons

Name	Gremlin	nGQL
vertex, node	vertex	vertex
edge, relationship	edge	edge
vertex type	label	tag
edge type	label	edge type
vertex id	vid	vid
edge id	eid	not support

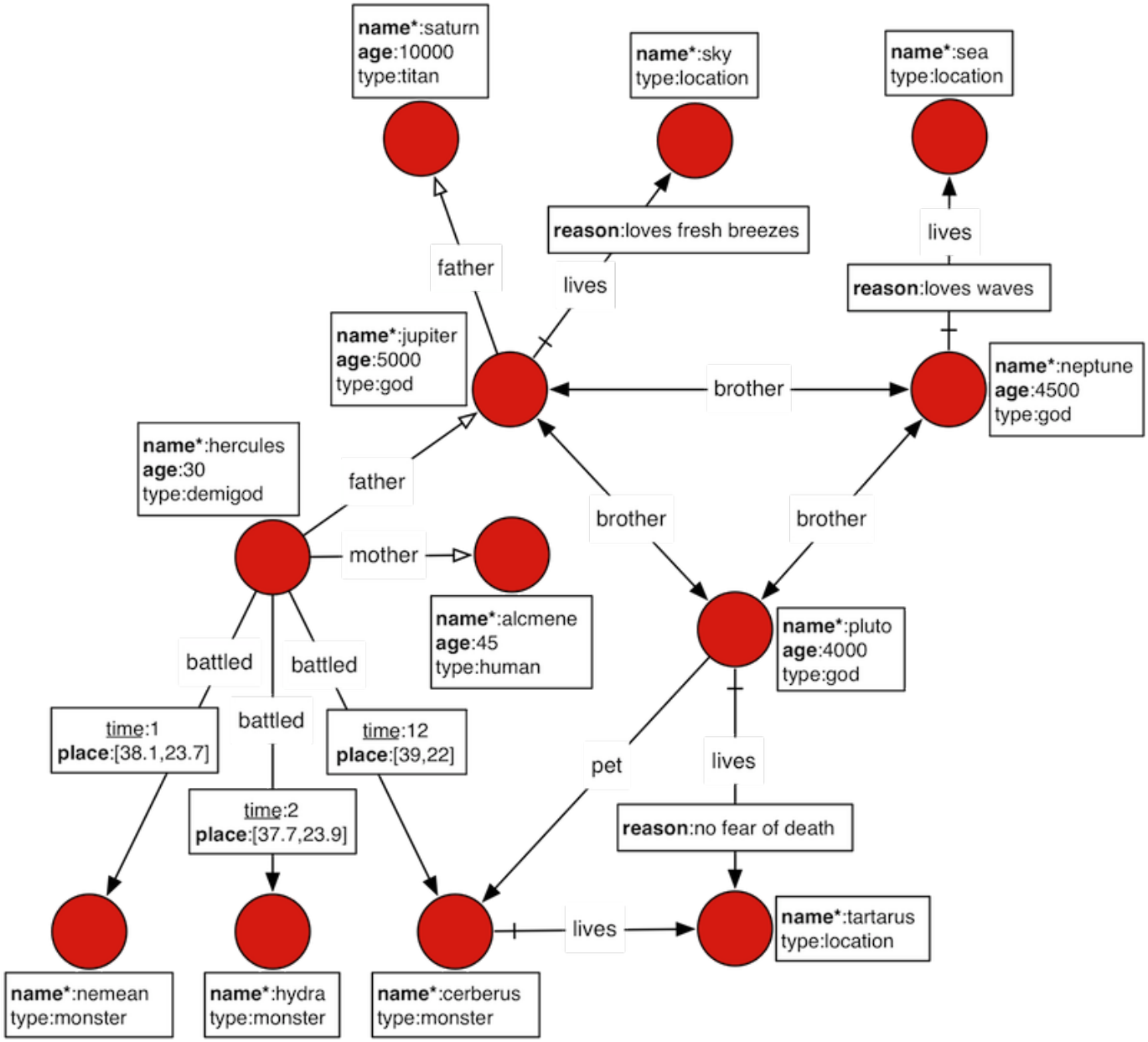
In Gremlin and nGQL, vertices and edges are identified with unique identifiers. In **Nebula Graph**, you can either specify identifiers or generate automatically with the hash or uuid function.

7.2.4 Basic Graph Operations

Name	Gremlin	nGQL
Create a new graph	g = TinkerGraph.open().traversal()	CREATE SPACE gods
Show vertices' types	g.V().label()	SHOW TAGS
Insert a vertex with a specified type	g.addV(String vertexLabel).property()	INSERT VERTEX `<tag_name>` (prop_name_list) VALUES `<vid>`: (prop_value_list)
Insert an edge with specified edge type	g.addE(String edgeLabel).from(v1).to(v2).property()	INSERT EDGE `<edge_name>` (`<prop_name_list>`) VALUES `<src_vid>` -> `<dst_vid>`: (`<prop_value_list>`)
Delete a vertex	g.V(`<vid>`).drop()	DELETE VERTEX `<vid>`
Delete an edge	g.E(`<vid>`).outE(`<type>`).where(otherV().is(`<vid>`))drop()	DELETE EDGE `<edge_type>` `<src_vid>` -> `<dst_vid>`
Update a vertex property	g.V(`<vid>`).property()	UPDATE VERTEX `<vid>` SET `<update_columns>`
Fetch vertices with ID	g.V(`<vid>`)	FETCH PROP ON `<tag_name>` `<vid>`
Fetch edges with ID	g.E(`<src_vid>` >> `<dst_vid>`)	FETCH PROP ON `<edge_name>` `<src_vid>` -> `<dst_vid>`
Query a vertex along specified edge type	g.V(`<vid>`).outE(`<edge>`)	GO FROM `<vid>` OVER `<edge>`
Query a vertex along specified edge type reversely	g.V(`<vid>`).in(`<edge>`)	GO FROM `<vid>` OVER `<edge>` REVERSELY
Query N hops along a specified edge	g.V(`<vid>`).repeat(out(`<edge>`)).times(N)	GO N STEPS FROM `<vid>` OVER `<edge>`
Find path between two vertices	g.V(`<vid>`).repeat(out()).until(`<vid>`).path()	FIND ALL PATH FROM `<vid>` TO `<vid>` OVER *

7.2.5 Example Queries

The examples in this section make extensive use of the toy graph distributed with Janus Graph called *The Graphs of Gods*. This graph is diagrammed below. The abstract data model is known as a Property Graph Model and this particular instance describes the relationships between the beings and places of the Roman pantheon.



- Insert data

```
# insert vertex
nebula> INSERT VERTEX character(name,age, type) VALUES hash("saturn"):("saturn", 10000, "titan"), hash("jupiter"):("jupiter", 5000, "god");

gremlin> saturn = g.addV("character").property(T.id, 1).property('name', 'saturn').property('age', 10000).property('type', 'titan').next();
==>v[1]
gremlin> jupiter = g.addV("character").property(T.id, 2).property('name', 'jupiter').property('age', 5000).property('type', 'god').next();
==>v[2]
gremlin> prometheus = g.addV("character").property(T.id, 31).property('name',  'prometheus').property('age', 1000).property('type', 'god').next();
==>v[31]
gremlin> jesus = g.addV("character")property(T.id, 32).property('name', 'jesus').property('age', 5000).property('type', 'god').next();
==>v[32]

# insert edge
nebula> INSERT EDGE father() VALUES hash("jupiter")->hash("saturn"):();
gremlin> g.addE("father").from(jupiter).to(saturn).property(T.id, 13);
==>e[13][2-father->1]
```

- Delete vertex

```
nebula> DELETE VERTEX hash("prometheus");
gremlin> g.V(prometheus).drop();
```

- Update vertex

```
nebula> UPDATE VERTEX hash("jesus") SET character.type = 'titan';
gremlin> g.V(jesus).property('age', 6000);
```

- Fetch data

```
nebula> FETCH PROP ON character hash("saturn");
=================================================
| character.name | character.age |character.type |
=================================================
| saturn         | 10000         |titan          |
-------------------------------------------------

gremlin> g.V(saturn).valueMap();
==>[name:[saturn],type:[titan],age:[10000]]
```

- Find the name of hercules's grandfather

```
nebula> LOOKUP ON character WHERE character.name == 'hercules' | \
     -> GO 2 STEPS FROM $-.VertexID OVER father YIELD $$.character.name;
=====================
| $$.character.name |
=====================
| saturn            |
---------------------

gremlin> g.V().hasLabel('character').has('name','hercules').out('father').out('father').values('name');
==>saturn
```

- Find the name of hercules's father

```
nebula> LOOKUP ON character WHERE character.name == 'hercules' | \
     -> GO FROM $-.VertexID OVER father YIELD $$.character.name;
=====================
| $$.character.name |
=====================
| jupiter           |
---------------------

gremlin> g.V().hasLabel('character').has('name','hercules').out('father').values('name');
==>jupiter
```

- Find the characters with age > 100

```
nebula> LOOKUP ON character WHERE character.age > 100 YIELD character.name;
=======================================
| VertexID             | character.name |
=======================================
| 6761447489613431910  | pluto          |
---------------------------------------
| -5860788569139907963 | neptune        |
---------------------------------------
| 4863977009196259577  | jupiter        |
---------------------------------------
| -4316810810681305233 | saturn         |
---------------------------------------

gremlin> g.V().hasLabel('character').has('age',gt(100)).values('name');
==>saturn
==>jupiter
==>neptune
==>pluto
```

- Find who are pluto's cohabitants

```
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | \
GO FROM $-.place OVER lives REVERSELY YIELD $$.character.name AS cohabitants;
===============
| cohabitants |
===============
| pluto       |
---------------
| cerberus    |
---------------

gremlin> g.V(pluto).out('lives').in('lives').values('name');
```

```
==>pluto
==>cerberus
```

- pluto can't be his own cohabitant

```
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | GO FROM $-.place OVER lives REVERSELY WHERE \
$$.character.name != "pluto" YIELD $$.character.name AS cohabitants;
===============
| cohabitants |
===============
| cerberus    |
---------------

gremlin> g.V(pluto).out('lives').in('lives').where(is(neq(pluto))).values('name');
==>cerberus
```

- Pluto's Brothers

```
# where do pluto's brothers live?

nebula> GO FROM hash("pluto") OVER brother YIELD brother._dst AS brother | \
GO FROM $-.brother OVER lives YIELD $$.location.name;
====================
| $$.location.name |
====================
| sky              |
--------------------
| sea              |
--------------------

gremlin> g.V(pluto).out('brother').out('lives').values('name');
==>sky
==>sea

# which brother lives in which place?

nebula> GO FROM hash("pluto") OVER brother YIELD brother._dst AS god | \
GO FROM $-.god OVER lives YIELD $^.character.name AS Brother, $$.location.name AS Habitations;
========================
| Brother | Habitations |
========================
| jupiter | sky         |
------------------------
| neptune | sea         |
------------------------

gremlin> g.V(pluto).out('brother').as('god').out('lives').as('place').select('god','place').by('name');
==>[god:jupiter, place:sky]
==>[god:neptune, place:sea]
```

7.2.6 Advance Queries

Graph Exploration

```
# Gremlin version
gremlin> Gremlin.version();
==>3.3.5

# Return all the vertices
gremlin> g.V();
==>v[1]
==>v[2]
...
nebula> # Coming soon

# Count all the vertices
gremlin> g.V().count();
==>12
nebula> # Coming soon

# Count the vertices and edges by label
gremlin> g.V().groupCount().by(label);
==>[character:9,location:3]
gremlin> g.E().groupCount().by(label);
==>[mother:1,lives:5,father:2,brother:6,battled:3,pet:1]
nebula> # Coming soon

# Return all edges
gremlin> g.E();
==>e[13][2-father->1]
==>e[14][2-lives->3]
...
nebula> # Coming soon

# Return vertices labels
gremlin> g.V().label().dedup();
==>character
```

```
==>location

nebula> SHOW TAGS;
==================
| ID | Name      |
==================
| 15 | character |
------------------
| 16 | location  |
------------------

# Return edge types
gremlin> g.E().label().dedup();
==>father
==>lives
...nebula> SHOW EDGES;
================
| ID | Name    |
================
| 17 | father  |
----------------
| 18 | brother |
----------------
...

# Return all vertices properties
gremlin> g.V().valueMap();
==>[name:[saturn],type:[titan],age:[10000]]
==>[name:[jupiter],type:[god],age:[5000]]
...
nebula> # Coming soon

# Return properties of vertices labeled character
gremlin> g.V().hasLabel('character').valueMap();
==>[name:[saturn],type:[titan],age:[10000]]
==>[name:[jupiter],type:[god],age:[5000]]
...
```

Traversing Edges

Name	Gremlin	nGQL
Out adjacent vertices to the vertex	out(\)	GO FROM \ OVER \
In adjacent vertices to the vertex	in(\)	GO FROM \ OVER \ REVERSELY
Both adjacent vertices of the vertex	both(\)	GO FROM \ OVER \ BIDIRECT

```
# Find the out adjacent vertices of a vertex along an edge
gremlin> g.V(jupiter).out('brother');
==>v[8]
==>v[5]
nebula> GO FROM hash("jupiter") OVER brother;
========================
| brother._dst         |
========================
| 6761447489613431910  |
------------------------
| -5860788569139907963 |
------------------------

# Find the in adjacent vertices of a vertex along an edge
gremlin> g.V(jupiter).in('brother');
==>v[5]
==>v[8]
nebula> GO FROM hash("jupiter") OVER brother REVERSELY;
========================
| brother._dst        |
========================
| 4863977009196259577 |
------------------------
| 4863977009196259577 |
------------------------

# Find the both adjacent vertices of a vertex along an edge
gremlin> g.V(jupiter).both('brother');
==>v[8]
==>v[5]
==>v[5]
==>v[8]
nebula> GO FROM hash("jupiter") OVER brother BIDIRECT;
========================
| brother._dst        |
========================
| 6761447489613431910 |
------------------------
| -5860788569139907963|
```

```
| 4863977009196259577 |
----------------------
| 4863977009196259577 |
----------------------

# Two hops out traverse
gremlin> g.V(hercules).out('father').out('lives');
==>v[3]
nebula> GO FROM hash("hercules") OVER father YIELD father._dst AS id | \
GO FROM $-.id OVER lives;
========================
| lives._dst           |
========================
| -1121386748834253737 |
------------------------
```

Has Filter Condition

Name	Gremlin	nGQL
Filter vertex via identifier	hasId(\)	FETCH PROP ON \
Filter vertex or edge via label, key and value	has(\, \, \)	LOOKUP \ | \ WHERE \

```
# Filter vertex with ID saturn
gremlin> g.V().hasId(saturn);
==>v[1]
nebula> FETCH PROP ON * hash("saturn");
================================================================================
| VertexID             | character.name | character.age | character.type |
================================================================================
| -4316810810681305233 | saturn         | 10000         | titan          |
--------------------------------------------------------------------------------

# Find for vertices with tag "character" and "name" attribute value "hercules"

gremlin> g.V().has('character','name','hercules').valueMap();
==>[name:[hercules],type:[demigod],age:[30]]
nebula> LOOKUP ON character WHERE character.name == 'hercules' YIELD character.name, character.age, character.type;
================================================================================
| VertexID            | character.name | character.age | character.type |
================================================================================
| 5976696804486077889 | hercules       | 30            | demigod        |
--------------------------------------------------------------------------------
```

Limiting Returned Results

Name	Gremlin	nGQL
Constrain the number of rows to return	limit()	LIMIT
Emit the last n-objects	tail()	ORDER BY \ DESC LIMIT
Skip n-objects	skip()	LIMIT \

```
# Find the first two records
gremlin> g.V().has('character','name','hercules').out('battled').limit(2);
==>v[9]
==>v[10]
nebula> GO FROM hash('hercules') OVER battled | LIMIT 2;
======================
| battled._dst       |
======================
| 530133512982221454 |
----------------------
| -695163537569412701 |
----------------------

# Find the last record
gremlin> g.V().has('character','name','hercules').out('battled').values('name').tail(1);
==>cerberus
nebula> GO FROM hash('hercules') OVER battled YIELD $$.character.name AS name | ORDER BY name | LIMIT 1;
============
| name     |
============
| cerberus |
------------

# Skip the first record and return one record
gremlin> g.V().has('character','name','hercules').out('battled').values('name').skip(1).limit(1);
==>hydra
```

```
nebula> GO FROM hash('hercules') OVER battled YIELD $$.character.name AS name | ORDER BY name | LIMIT 1,1;
=========
| name  |
=========
| hydra |
---------
```

Finding Path

Name	Gremlin	nGQL
All path	path()	FIND ALL PATH
Exclude cycles path	simplePath()	\
Only cycles path	cyclicPath()	\
Shortest path	\	FIND SHORTEST PATH

NOTE: Nebula Graph requires the source vertex and the destination vertex to find path while Gremlin only needs the source vertex.

```
# Find path from vertex pluto to the out adjacent vertices
gremlin> g.V().hasLabel('character').has('name','pluto').out().path();
==>[v[8],v[12]]
==>[v[8],v[2]]
==>[v[8],v[5]]
==>[v[8],v[11]]

# Find the shortest path from vertex pluto to vertex jupiter
nebula> LOOKUP ON character WHERE character.name== "pluto" YIELD character.name AS name | \
    FIND SHORTEST PATH FROM $-.VertexID TO hash("jupiter") OVER *;
============================================================
| _path_                |
============================================================
| 6761447489613431910 <brother,0> 4863977009196259577
------------------------------------------------------------
```

Traversing N Hops

Name	Gremlin	nGQL
Loop over a traversal	repeat()	N STEPS
Times the traverser has gone through a loop	times()	N STEPS
Specify when to end the loop	until()	\
Specify when to collect data	emit()	\

```
# Find vertex pluto's out adjacent neighbors
gremlin> g.V().hasLabel('character').has('name','pluto').repeat(out()).times(1);
==>v[12]
==>v[2]
==>v[5]
==>v[11]
nebula> LOOKUP ON character WHERE character.name== "pluto" YIELD character.name AS name | \
    GO FROM $-.VertexID OVER *;
=================================================================================================================
| father._dst | brother._dst         | lives._dst          | mother._dst | pet._dst            | battled._dst |
=================================================================================================================
| 0           | -586078569139907963 | 0                   | 0           | 0                   | 0            |
-----------------------------------------------------------------------------------------------------------------
| 0           | 4863977009196259577  | 0                   | 0           | 0                   | 0            |
-----------------------------------------------------------------------------------------------------------------
| 0           | 0                   | -4331657707562925133 | 0           | 0                   | 0            |
-----------------------------------------------------------------------------------------------------------------
| 0           | 0                   | 0                   | 0           | 4594048193862126013 | 0            |
-----------------------------------------------------------------------------------------------------------------

# Find path between vertex hercules and vertex cerberus
# Stop traversing when the destination vertex is cerberus
gremlin> g.V().hasLabel('character').has('name','hercules').repeat(out()).until(has('name', 'cerberus')).path();
==>[v[6],v[11]]
==>[v[6],v[2],v[8],v[11]]
==>[v[6],v[2],v[5],v[8],v[11]]
...
nebula> # Coming soon
```

```
# Find path sourcing from vertex hercules
# And the destination vertex type is character
gremlin> g.V().hasLabel('character').has('name','hercules').repeat(out()).emit(hasLabel('character')).path();
==>[v[6],v[7]]
==>[v[6],v[2]]
==>[v[6],v[9]]
==>[v[6],v[10]]
...
nebula> # Coming soon

# Find shortest path between pluto and saturn over any edge
# And the deepest loop is 3
gremlin> g.V('pluto').repeat(out().simplePath()).until(hasId('saturn').and().loops().is(lte(3))).hasId('saturn').path();
nebula> FIND SHORTEST PATH FROM hash('pluto') TO hash('saturn') OVER * UPTO 3 STEPS;
==========================================================================================
| _path_                                                                                 |
==========================================================================================
| 6761447489613431910 <brother,0> 4863977009196259577 <father,0> -4316810810681305233
------------------------------------------------------------------------------------------
```

Ordering Results

Name	Gremlin	nGQL
Order the items increasingly	order().by()	ORDER BY
Order the items decreasingly	order().by(decr)	ORDER BY DESC
Randomize the records order	order().by(shuffle)	\

```
# Find pluto's brother and order by age decreasingly.
gremlin> g.V(pluto).out('brother').order().by('age', decr).valueMap();
==>[name:[jupiter],type:[god],age:[5000]]
==>[name:[neptune],type:[god],age:[4500]]
nebula> GO FROM hash('pluto') OVER brother YIELD $$.character.name AS Name, $$.character.age as Age | ORDER BY Age DESC;
==================
| Name    | Age  |
==================
| jupiter | 5000 |
------------------
| neptune | 4500 |
------------------
```

Group By

Name	Gremlin	nGQL
Group by items	group().by()	GROUP BY
Remove repeated items	dedup()	DISTINCT
Group by items and count	groupCount()	GROUP BY COUNT

NOTE: The GROUP BY function can only be applied in the YIELD clause.

```
# Group vertices by label then count
gremlin> g.V().group().by(label).by(count());
==>[character:9,location:3]
nebula> # Coming soon

# Find vertex jupiter's out adjacency vertices, group by name, then count
gremlin> g.V(jupiter).out().group().by('name').by(count());
==>[sky:1,saturn:1,neptune:1,pluto:1]
nebula> GO FROM hash('jupiter') OVER * YIELD $$.character.name AS Name, $$.character.age as Age, $$.location.name | \
GROUP BY $-.Name YIELD $-.Name, COUNT(*);
======================
| $-.Name | COUNT(*) |
======================
|         | 1        |
----------------------
| pluto   | 1        |
----------------------
| saturn  | 1        |
----------------------
| neptune | 1        |
----------------------

# Find the distinct destination vertices sourcing from vertex jupiter
gremlin> g.V(jupiter).out().hasLabel('character').dedup();
```

```
==>v[1]
==>v[8]
==>v[5]
nebula> GO FROM hash('jupiter') OVER * YIELD DISTINCT $$.character.name, $$.character.age, $$.location.name;
=============================================================
| $$.character.name | $$.character.age | $$.location.name |
=============================================================
| pluto             | 4000             |                  |
-------------------------------------------------------------
| neptune           | 4500             |                  |
-------------------------------------------------------------
| saturn            | 10000            |                  |
-------------------------------------------------------------
|                   | 0                | sky              |
-------------------------------------------------------------
```

Where Filter Condition

Name	Gremlin	nGQL
Where filter condition	where()	WHERE

Predicates comparison:

Name	Gremlin	nGQL
Equal to	eq(object)	==
Not equal to	neq(object)	!=
Less than	lt(number)	<
Less than or equal to	lte(number)	<=
Greater than	gt(number)	>
Greater than or equal to	gte(number)	>=
Whether a value is within the array	within(objects...)	udf_is_in()

```
gremlin> eq(2).test(3);
==>false
nebula> YIELD 3 == 2;
==========
| (3==2) |
==========
| false  |
----------

gremlin> within('a','b','c').test('d');
==>false
nebula> YIELD udf_is_in('d', 'a', 'b', 'c');
======================
| udf_is_in(d,a,b,c) |
======================
| false              |
----------------------
```

```
# Find pluto's co-habitants and exclude himself
gremlin> g.V(pluto).out('lives').in('lives').where(is(neq(pluto))).values('name');
==>cerberus
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | GO FROM $-.place OVER lives REVERSELY WHERE \
$$.character.name != "pluto" YIELD $$.character.name AS cohabitants;
===============
| cohabitants |
===============
| cerberus    |
---------------
```

Logical Operators

Name	Gremlin	nGQL
Is	is()	==
Not	not()	!=
And	and()	AND
Or	or()	OR

```
# Find age greater than or equal to 30
gremlin> g.V().values('age').is(gte(30));
==>10000
==>5000
==>4500
==>30
==>45
==>4000
nebula> LOOKUP ON character WHERE character.age >= 30 YIELD character.age;
=======================================
| VertexID             | character.age |
=======================================
| -4316810810681305233 | 10000         |
---------------------------------------
| 4863977009196259577  | 5000          |
---------------------------------------
| -5860788569139907963 | 4500          |
---------------------------------------
| 5976696804486077889  | 30            |
---------------------------------------
| -6780323075177699500 | 45            |
---------------------------------------
| 6761447489613431910  | 4000          |
---------------------------------------

# Find character with name pluto and age 4000
gremlin> g.V().has('name','pluto').and().has('age',4000);
==>v[8]
nebula> LOOKUP ON character WHERE character.name == 'pluto' AND character.age == 4000;
======================
| VertexID           |
======================
| 6761447489613431910 |
----------------------

# Logical not
gremlin> g.V().has('name','pluto').out('brother').not(values('name').is('neptune')).values('name');
==>jupiter
nebula> LOOKUP ON character WHERE character.name == 'pluto' YIELD character.name AS name | \
GO FROM $-.VertexID OVER brother WHERE $$.character.name != 'neptune' YIELD $$.character.name;
====================
| $$.character.name |
====================
| jupiter          |
--------------------
```

Statistical Operations

Name	Gremlin	nGQL
Sum	sum()	SUM()
Max	max()	MAX()
Min	min()	MIN()
Mean	mean()	AVG()

Nebula Graph statistical operations must be applied with `GROUP BY`.

```
# Calculate the sum of ages of all characters
gremlin> g.V().hasLabel('character').values('age').sum();
==>23595
nebula> # Coming soon

# Calculate the sum of the out brother edges of all characters
gremlin> g.V().hasLabel('character').map(outE('brother').count()).sum();
==>6
nebula> # Coming soon
```

```
# Return the max age of all characters
gremlin> g.V().hasLabel('character').values('age').max();
==>10000
nebula> # Coming soon
```

Selecting and Filtering Paths

```
# Select the results of steps 1 and 3 from the path as the final result
gremlin> g.V(pluto).as('a').out().as('b').out().as('c').select('a','c');
==>[a:v[8],c:v[3]]
==>[a:v[8],c:v[1]]
...
nebula> # Coming soon

# Specify dimensions via by()
gremlin> g.V(pluto).as('a').out().as('b').out().as('c').select('a','c').by('name');
==>[a:pluto,c:sky]
==>[a:pluto,c:saturn]
...
nebula> # Coming soon

# Selects the specified key value from the map
gremlin> g.V().valueMap().select('name').dedup();
==>[saturn]
==>[jupiter]
...
nebula> # Coming soon
```

Branches

```
# Traverse all vertices with label 'character'
# If name is 'jupiter', return the age property
# Else return the name property
gremlin> g.V().hasLabel('character').choose(values('name')).option('jupiter', values('age')).option(none, values('name'));
==>saturn
==>5000
==>neptune
...

# Lambda
gremlin> g.V().branch {it.get().value('name')}.option('jupiter', values('age')).option(none, values('name'));
==>saturn
==>5000
...

# Traversal
gremlin> g.V().branch(values('name')).option('jupiter', values('age')).option(none, values('name'));
==>saturn
==>5000

# Branch
gremlin> g.V().choose(has('name','jupiter'),values('age'),values('name'));
==>saturn
==>5000

# Group based on if then
gremlin> g.V().hasLabel("character").groupCount().by(values("age").choose(
           is(lt(40)),constant("young"),
            choose(is(lt(4500)),
                   constant("old"),
                  constant("very old"))));
==>[young:4,old:2,very old:3]
```

Similar function is yet to be supported in **Nebula Graph**.

Coalesce

The `coalesce()` step evaluates the provided traversals in order and returns the first traversal that emits at least one element.

The `optional()` step returns the result of the specified traversal if it yields a result else it returns the calling element, i.e. the identity().

The `union()` step supports the merging of the results of an arbitrary number of traversals.

```
# If type is monster, return type. Else return 'Not a monster'.
gremlin> g.V(pluto).coalesce(has('type','monster').values('type'),constant("Not a monster"));
==>Not a monster

# Find the following edges and adjacent vertices of jupiter in order, and stop when finding one
# 1. Edge brother out adjacent vertices
```

```
# 2. Edge father out adjacent vertices
# 3. Edge father in adjacent vertices
gremlin> g.V(jupiter).coalesce(outE('brother'), outE('father'), inE('father')).inV().path().by('name').by(label);
==>[jupiter,brother,pluto]
==>[jupiter,brother,neptune]

# Find pluto's father, if there is not any then return pluto himself
gremlin> g.V(pluto).optional(out('father')).valueMap();
==>[name:[pluto],type:[god],age:[4000]]

# Find pluto's father and brother, union the results then return the paths
gremlin> g.V(pluto).union(out('father'),both('brother')).path();
==>[v[8],v[2]]
==>[v[8],v[5]]
```

Similar function is yet to be supported in **Nebula Graph**.

Aggregating and Unfolding Results

```
# Collect results of the first step into set x
# Note: This operation doesn't affect subsequent results
gremlin> g.V(pluto).out().aggregate('x');
==>v[12]
==>v[2]
...

# Specify the aggregation dimensions via by ()
gremlin> g.V(pluto).out().aggregate('x').by('name').cap('x');
==>[tartarus,jupiter,neptune,cerberus]

# Find pluto's 2 hop out adjacent neighbors
# Collect the results in set x
# Show the neighbors' name
gremlin> g.V(pluto).out().aggregate('x').out().aggregate('x').cap('x').unfold().values('name');
==>tartarus
==>tartarus
...
```

Similar function is yet to be supported in **Nebula Graph**.

Matching Patterns

The `match()` step provides a more declarative form of graph querying based on the notion of pattern matching. With match(), the user provides a collection of "traversal fragments," called patterns, that have variables defined that must hold true throughout the duration of the match().

```
# Matching each vertex with the following pattern. If pattern is met, return map<String, Object>, els filter it.
# Pattern 1: a is jupiter's son
# Pattern 2: b is jupiter
# Pattern 3: c is jupiter's brother, whose age is 4000
gremlin> g.V().match(__.as('a').out('father').has('name', 'jupiter').as('b'), __.as('b').in('brother').has('age', 4000).as('c'));
==>[a:v[6],b:v[2],c:v[8]]

# match() can be applied with  select() to select partial results from Map <String, Object>
gremlin> g.V().match(__.as('a').out('father').has('name', 'jupiter').as('b'), __.as('b').in('brother').has('age', 4000).as('c')).select('a', 'c').by('name');
==>[a:hercules,c:pluto]

# match () can be applied with where () to filter the results
gremlin> g.V().match(__.as('a').out('father').has('name', 'jupiter').as('b'), __.as('b').in('brother').has('age', 4000).as('c')).where('a',
neq('c')).select('a', 'c').by('name');
==>[a:hercules,c:pluto]
```

Random filtering

The `sample()` step accepts an integer value and samples the maximum number of the specified results randomly from the previous traverser.

The `coin()` step can randomly filter out a traverser with the given probability. You give coin a value indicating how biased the toss should be.

```
# Randomly select 2 out edges from all vertices
gremlin> g.V().outE().sample(2);
==>e[15][2-brother->5]
==>e[18][5-brother->2]

# Pick 3 names randomly from all vertices
gremlin> g.V().values('name').sample(3);
==>hercules
==>sea
```

```
==>jupiter

# Pick 3 randomly from all characters based on age
gremlin> g.V().hasLabel('character').sample(3).by('age');
==>v[1]
==>v[2]
==>v[6]

# Applied with local to do random walk
# Starting from pluto, conduct random walk 3 times
gremlin> g.V(pluto).repeat(local(bothE().sample(1).otherV())).times(3).path();
==>[v[8],e[26][8-brother->5],v[5],e[18][5-brother->2],v[2],e[13][2-father->1],v[1]]

# Filter each vertex with a probability of 0.5
gremlin> g.V().coin(0.5);
==>v[1]
==>v[2]
...

# Return the name attribute of all vertices labeled location, otherwise return not a location
gremlin> g.V().choose(hasLabel('location'), values('name'), constant('not a location'));
==>not a location
==>not a location
==>sky
...
```

Sack

A traverser that contains a local data structure is called a "sack". The `sack()` step is used to read and write sacks. Each sack of each traverser is created with `withSack()`.

```
# Defines a Gremlin sack with a value of one and return values in the sack
gremlin> g.withSack(1).V().sack();
==>1
==>1
...
```

Barrier

The `barrier()` step turns the lazy traversal pipeline into a bulk-synchronous pipeline. It's useful when everything prior to barrier() needs to be executed before moving onto the steps after the barrier().

```
# Calculate the Eigenvector Centrality with barrier
# Including groupCount and cap, sorted in descending order
gremlin> g.V().repeat(both().groupCount('m')).times(5).cap('m').order(local).by(values, decr);
```

Local

A GraphTraversal operates on a continuous stream of objects. In many situations, it is important to operate on a single element within that stream. To do such object-local traversal computations, `local()` step exists.

```
# Without local()
gremlin> g.V().hasLabel('character').as('character').properties('age').order().by(value,decr).limit(2).value().as('age').select('character',
'age').by('name').by();
==>[character:saturn,age:10000]
==>[character:jupiter,age:5000]

# With local()
gremlin> g.V().hasLabel('character').as('character').local(properties('age').order().by(value).limit(2)).value().as('age').select('character',
'age').by('name').by()
==>[character:saturn,age:10000]
==>[character:jupiter,age:5000]
==>[character:neptune,age:4500]
==>[character:hercules,age:30]
...

# Return the property map of monster
gremlin> g.V()hasLabel('character').has('type', 'type').propertyMap();
==>[name:[vp[name->nemean]],type:[vp[type->monster]],age:[vp[age->20]]]
==>[name:[vp[name->hydra]],type:[vp[type->monster]],age:[vp[age->0]]]
==>[name:[vp[name->cerberus]],type:[vp[type->monster]],age:[vp[age->0]]]

# Find number of monster
gremlin> g.V()hasLabel('character').has('type', 'monster').propertyMap().count(local);
==>3
==>3
==>3

# Find the max vertices number labeled tha same tag
gremlin> g.V().groupCount().by(label).select(values).max(local);
==>9
```

```
# List the first attribute of all vertices
gremlin> g.V().valueMap().limit(local, 1);
==>[name:[saturn]]
==>[name:[jupiter]]
==>[name:[sky]]
...

# Without local
gremlin> g.V().valueMap().limit(1);
==>[name:[saturn],type:[titan],age:[10000]]

# All vertices as a set, sample 2 from it
gremlin> g.V().fold().sample(local,2);
==>[v[8],v[1]]
```

Statistics and Analysis

Gremlin provides two steps for statistics and analysis of the executed query statements:

- The `explain()` step will return a TraversalExplanation. A traversal explanation details how the traversal (prior to explain()) will be compiled given the registered traversal strategies.
- The `profile()` step allows developers to profile their traversals to determine statistical information like step runtime, counts, etc.

Last update: April 8, 2021

7.3 Comparison Between SQL and nGQL

7.3.1 Conceptual Comparisons

Items	SQL	nGQL
vertex	\	vertex
edge	\	edge
vertex type	\	tag
edge type	\	edge type
vertex identifier	primary key	vid
edge identifier	composite primary key	src, dst, rank
column	column	properties of vertices or edges
row	row	one vertex or edge

7.3.2 Syntax Comparisons

Data Definition Language (DDL)

Data Definition Language (DDL) can be used to define a database schema. DDL statements create and modify the structure of a database.

Items	SQL	nGQL
Create (graph) database	CREATE DATABASE `<database_name>`	CREATE SPACE `<space_name>`
Show (graph) database	SHOW DATABASES	SHOW SPACES
Use (graph) database	USE `<database_name>`	USE `<space_name>`
Drop (graph) database	DROP DATABASE `<database_name>`	DROP SPACE `<space_name>`
Alter (graph) database	ALTER DATABASE `<database_name>` alter_option	\
Create tags/edges	\	CREATE TAG | EDGE `<tag_name>`
Create a table	CREATE TABLE `<tbl_name>` (create_definition,...)	\
Show columns	SHOW COLUMNS FROM `<tbl_name>`	\
Show tags/edges	\	SHOW TAGS | EDGES
Describe tags/edge	\	DESCRIBE TAG | EDGE `<tag_name | edge_name>`
Alter a tag/edge	\	ALTER TAG | EDGE `<tag_name | edge_name>`
Alter a table	ALTER TABLE `<tbl_name>`	\

INDEX

Items	SQL	nGQL
Create index	CREATE INDEX	CREATE {TAG | EDGE} INDEX
Drop index	DROP INDEX	DROP {TAG | EDGE} INDEX
Show index	SHOW INDEX FROM	SHOW {TAG | EDGE} INDEXES
Rebuild index	ANALYZE TABLE	REBUILD {TAG | EDGE} INDEX `<index_name>` [OFFLINE]

Data Manipulation Language (DML)

Data Manipulation Language (DML) is used to manipulate data in a database.

Items	SQL	nGQL
Insert data	INSERT IGNORE INTO `<tbl_name>` [(col_name [, col_name] ...)] {VALUES | VALUE} [(value_list) [, (value_list)]	INSERT VERTEX `<tag_name>` (prop_name_list[, prop_name_list]) {VALUES | VALUE} vid: (prop_value_list[, prop_value_list]) INSERT EDGE `<edge_name>` (`<prop_name_list>`) VALUES | VALUE `<src_vid>` -> `<dst_vid>` [`@<rank>`] : (`<prop_value_list>`)
Query data	SELECT	GO, FETCH
Update data	UPDATE `<tbl_name>` SET field1=new-value1, field2=new-value2 [WHERE Clause]	UPDATE VERTEX `<vid>` SET `<update_columns>` [WHEN `<condition>`] UPDATE EDGE `<edge>` SET `<update_columns>` [WHEN `<condition>`]
Delete data	DELETE FROM `<tbl_name>` [WHERE Clause]	DELETE EDGE `<edge_type>` `<vid>` -> `<vid>` [`@<rank>`] [, `<vid>` -> `<vid>` ...] DELETE VERTEX `<vid_list>`
Join data	JOIN	|

Data Query Language (DQL)

Data Query Language (DQL) statements are used for performing queries on the data. This section shows how you can query data with SQL statements and nGQL statements.

```
SELECT
 [DISTINCT]
 select_expr [, select_expr] ...
 [FROM table_references]
 [WHERE where_condition]
 [GROUP BY {col_name | expr | position}]
 [HAVING  where_condition]
 [ORDER BY {col_name | expr | position} [ASC | DESC]]
```

```
GO [[<M> TO] <N> STEPS ] FROM <node_list>
 OVER <edge_type_list> [REVERSELY] [BIDIRECT]
 [WHERE where_condition]
 [YIELD [DISTINCT] <return_list>]
 [| ORDER BY <expression> [ASC | DESC]]
 [| LIMIT [<offset_value>,] <number_rows>]
 [| GROUP BY {col_name | expr | position} YIELD <col_name>]

<node_list>
   | <vid> [, <vid> ...]
   | $-.id

<edge_type_list>
   edge_type [, edge_type ...]

<return_list>
    <col_name> [AS <col_alias>] [, <col_name> [AS <col_alias>] ...]
```

Data Control Language (DCL)

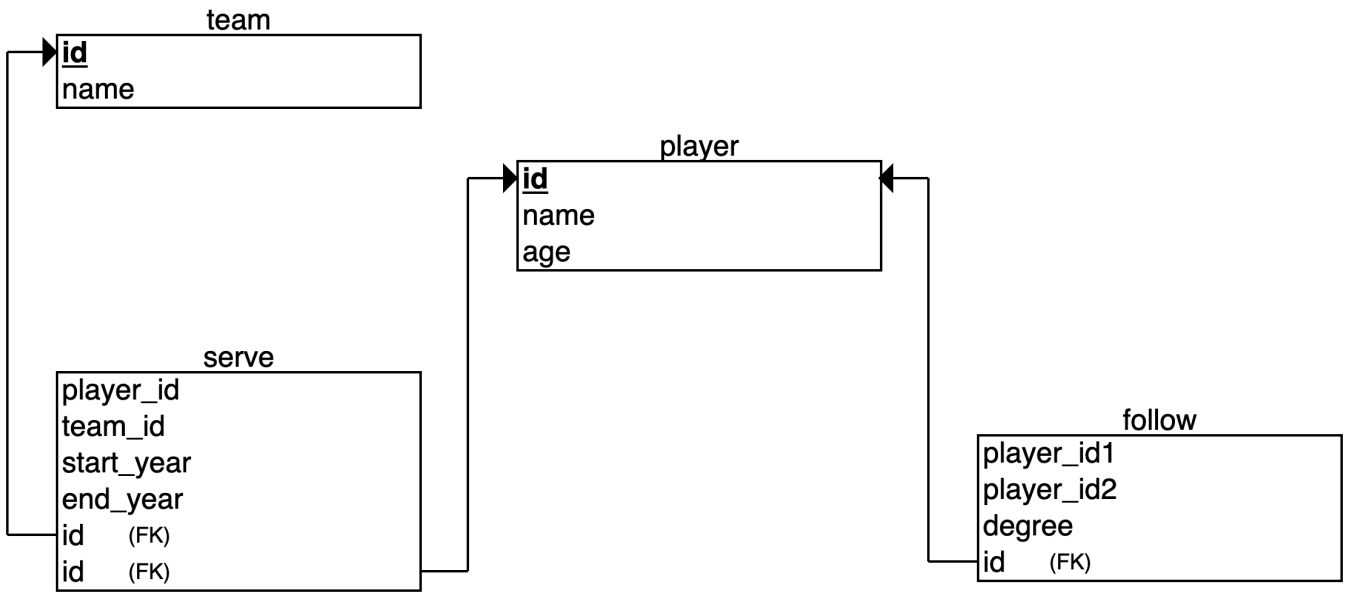
Data Control Language (DCL) includes commands such as `GRANT` and `REVOKE` that mainly deals with the rights, permissions, and other controls of the database system.

Items	SQL	nGQL
Create user	CREATE USER	CREATE USER
Drop user	DROP USER	DROP USER
Change password	SET PASSWORD	CHANGE PASSWORD
Grant privilege	GRANT `<priv_type>` ON [object_type] TO `<user>`	GRANT ROLE `<role_type>` ON `<space>` TO `<user>`
Revoke privilege	REVOKE `<priv_type>` ON [object_type] TO `<user>`	REVOKE ROLE `<role_type>` ON `<space>` FROM `<user>`

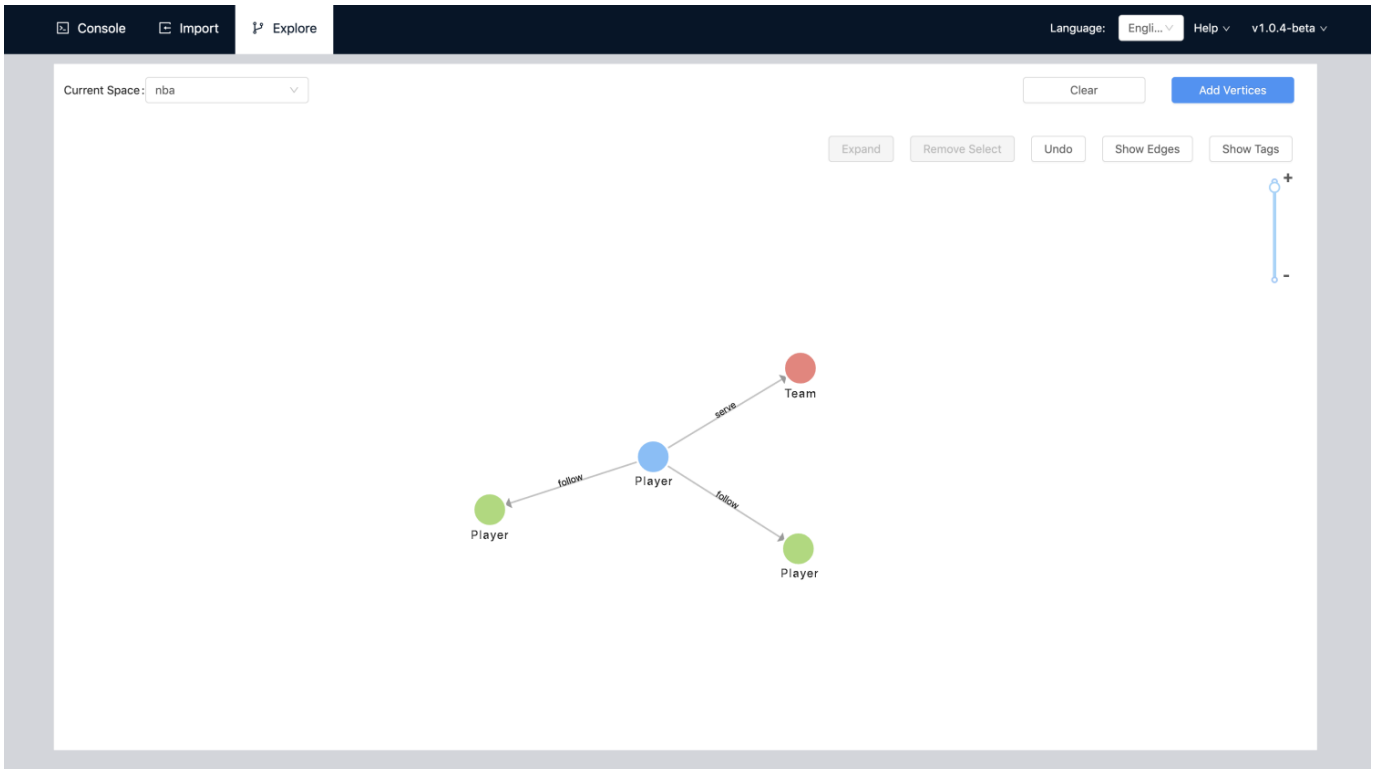
7.3.3 Data Model

The queries are based on the data model below:

MySQL



Nebula Graph



7.3.4 CRUD

This section describes how to create (C), read (R), update (U), and delete (D) data with SQL and nGQL statements.

Inserting Data

```
mysql> INSERT INTO player VALUES (100, 'Tim Duncan', 42);

nebula> INSERT VERTEX player(name, age) VALUES 100: ('Tim Duncan', 42);
```

Querying Data

Find the player whose id is 100 and output the `name` property:

```
mysql> SELECT player.name FROM player WHERE player.id = 100;

nebula> FETCH PROP ON player 100 YIELD player.name;
```

Updating Data

```
mysql> UPDATE player SET name = 'Tim';

nebula> UPDATE VERTEX 100 SET player.name = "Tim";
```

Deleting Data

```
mysql> DELETE FROM player WHERE name = 'Tim';

nebula> DELETE VERTEX 121;
nebula> DELETE EDGE follow 100 -> 200;
```

7.3.5 Sample Queries

Query 1

Find players who are younger than 36.

```
mysql> SELECT player.name
FROM player
WHERE player.age < 36;
```

The query in nGQL is a bit different because you must create an index before filtering a property. For more information, see Index Doc.

```
nebula> CREATE TAG INDEX player_age ON player(age);
nebula> REBUILD TAG INDEX player_age OFFLINE;
nebula> LOOKUP ON player WHERE player.age < 36;
```

Query 2

Find Tim Duncan and list all the teams that he served.

```
mysql> SELECT a.id, a.name, c.name
FROM player a
JOIN serve b ON a.id=b.player_id
JOIN team c ON c.id=b.team_id
WHERE a.name = 'Tim Duncan';
```

```
nebula> CREATE TAG INDEX player_name ON player(name);
nebula> REBUILD TAG INDEX player_name OFFLINE;
nebula> LOOKUP ON player WHERE player.name == 'Tim Duncan' YIELD player.name AS name | GO FROM $-.VertexID OVER serve YIELD $-.name, $$.team.name;
```

Query 3

Find Tim Duncan's teammates.

```
mysql> SELECT a.id, a.name, c.name
FROM player a
JOIN serve b ON a.id=b.player_id
JOIN team c ON c.id=b.team_id
WHERE c.name IN (SELECT c.name
FROM player a
JOIN serve b ON a.id=b.player_id
JOIN team c ON c.id=b.team_id
WHERE a.name = 'Tim Duncan');
```

In nGQL we use pipes to pass the output of the previous statement as the input for the next statement.

```
nebula> GO FROM 100 OVER serve YIELD serve._dst AS Team | GO FROM $-.Team OVER serve REVERSELY YIELD $$.player.name;
```

Last update: April 8, 2021

7.4 Vertex Identifier and Partition

This document provides some introductions to vertex identifier (`VID` for short) and partition.

In **Nebula Graph**, vertices are identified with vertex identifiers (i.e. `VID`s). When inserting a vertex, you must specify a `VID` for it. You can generate `VID`s either with your own application or with the hash function provided by **Nebula Graph**.

`VID`s must be unique in a graph space. That is, in the same graph space, vertices with the same `VID` are considered as the same vertex. `VID`s in different graph spaces are independent of each other. In addition, one `VID` can have multiple `TAG`s.

When inserting data into **Nebula Graph**, vertices and edges are distributed across different partitions. And the partitions are located on different machines. If you want some certain vertices to locate on the same partition (i.e. on the same machine), you can control the generation of the `VID`s by using the following formula.

The relation between `VID` and partition is:

```
VID mod partition_number = partition ID + 1
```

In the preceding formula,

- `mod` is the modulo operation.
- `partition_number` is the number of partition for the graph space where the `VID` is located, namely the value of `partition_num` in the CREATE SPACE statement.
- `partition ID` is the ID for the partition where the `VID` is located.

For example, if there are 100 partitions, the vertices with `VID` 1, 101, 1001 will be stored on the same partition.

In addition, the correspondence between the `partition ID` and the machines are random. Therefore, you can't assume that any two partitions are located on the same machine.

Last update: April 8, 2021

https://docs.nebula-graph.io