

Nebula Graph Database Manual

master

Min Wu, Yao Zhou, Cooper Liang

2021 Vesoft Inc.

Table of contents

1. Welcome to Nebula Graph 2.5.0 Documentation	5
1.1 Getting started	5
1.2 Other Sources	5
1.3 Graphic Illustrations	5
1.4 Modify errors	6
2. Introduction	7
2.1 What is Nebula Graph	7
2.2 Data modeling	11
2.3 Path types	13
2.4 VID	15
2.5 Nebula Graph architecture	17
3. Quick start	33
3.1 Quick start workflow	33
3.2 Manage Nebula Graph services	34
3.3 Connect to Nebula Graph	38
3.4 Nebula Graph CRUD	43
4. nGQL guide	53
4.1 nGQL overview	53
4.2 Data types	69
4.3 Variables and composite queries	86
4.4 Operators	91
4.5 Functions and expressions	104
4.6 General queries statements	127
4.7 Clauses and options	169
4.8 Space statements	191
4.9 Tag statements	198
4.10 Edge type statements	204
4.11 Vertex statements	210
4.12 Edge statements	217
4.13 Native index statements	224
4.14 Full-text index statements	234
4.15 Subgraph and path	244
4.16 Query tuning statements	251
4.17 Operation and maintenance statements	254

5. Deployment and installation	258
5.1 Prepare resources for compiling, installing, and running Nebula Graph	258
5.2 Compile and install Nebula Graph	266
5.3 Deploy Nebula Graph cluster	271
5.4 Upgrade Nebula Graph to v2.0.0	273
5.5 Uninstall Nebula Graph	279
6. Configurations and logs	281
6.1 Configurations	281
6.2 Log management	296
7. Monitor and metrics	298
7.1 Query Nebula Graph metrics	298
8. Data security	300
8.1 Authentication and authorization	300
8.2 Backup and restore data with snapshots	305
9. Service Tuning	307
9.1 Compaction	307
9.2 Storage load balance	309
10. Nebula Graph Dashboard	312
10.1 What is Nebula Graph Dashboard	312
10.2 Deploy Dashboard	313
10.3 Connect Dashboard	317
10.4 Dashboard	320
10.5 Metrics	324
11. Nebula Spark Connector	327
11.1 Use cases	327
11.2 Benefits	327
12. Nebula Flink Connector	328
12.1 Use cases	328
13. Contribution	329
13.1 How to Contribute	329
14. FAQ	332
14.1 FAQ	332
15. Appendix	337
15.1 VID	337
15.2 Graph modeling	338
15.3 System modeling	339
15.4 About Raft	340
15.5 Partition ID	341

15.6 Version Description	342
15.7 Comments	343
15.8 Identifier Case Sensitivity	344
15.9 Keywords and Reserved Words	345

1. Welcome to Nebula Graph 2.5.0 Documentation

 This manual is revised on 2021-8-27, with GitHub commit [7f822842d](#).

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 (relationships) with millisecond latency.

1.1 Getting started

- [What is Nebula Graph](#)
- [Quick start workflow](#)
- [Configuration](#)
- [FAQ](#)
- [Ecosystem Tools](#)

1.2 Other Sources

- [Nebula Graph Homepage](#)
- [Release note](#)
- [Forum](#)
- [Blog](#)
- [Video](#)
- [Chinese Docs](#)

1.3 Graphic Illustrations

 **Note**

Additional information or operation-related notes.

 **Caution**

Cautions that need strict observation. If not, systematic breakdown, data loss, and security issues may happen.

 **Danger**

Operations that may cause danger. If not observed, systematic breakdown, data loss, and security issues will happen.

 **Performance**

Operations that merit attention as for performance enhancement.

FAQ

Common questions.

Compatibility

The compatibility between nGQL and openCypher, or between the current version of nGQL and its prior ones.

Enterpriseonly

Differences between the Nebula Graph Open Source and Enterprise editions.

1.4 Modify errors

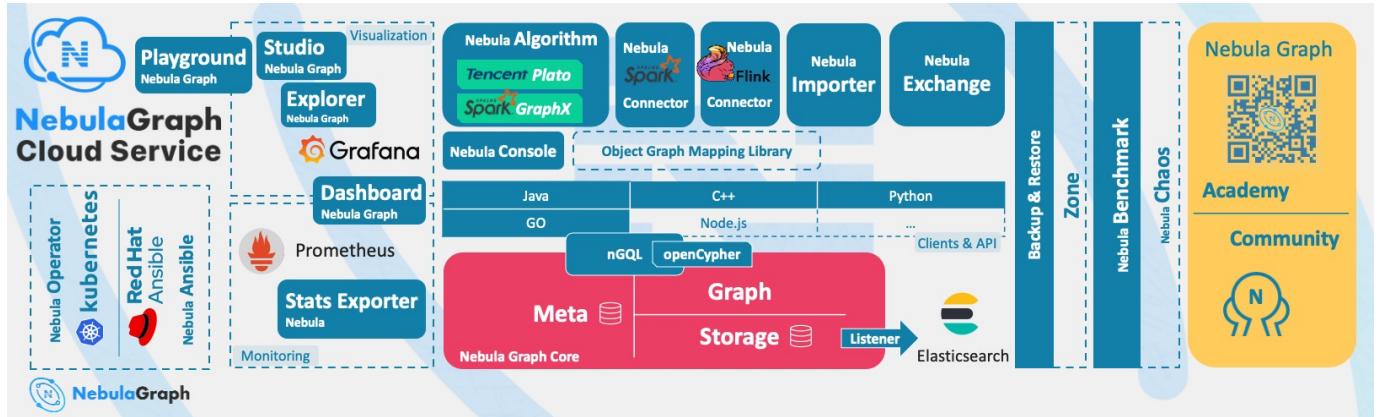
This Nebula Graph manual is written in the Markdown language. Users can click the pencil sign on the upper right side of each document title and modify errors.

Last update: August 24, 2021

2. Introduction

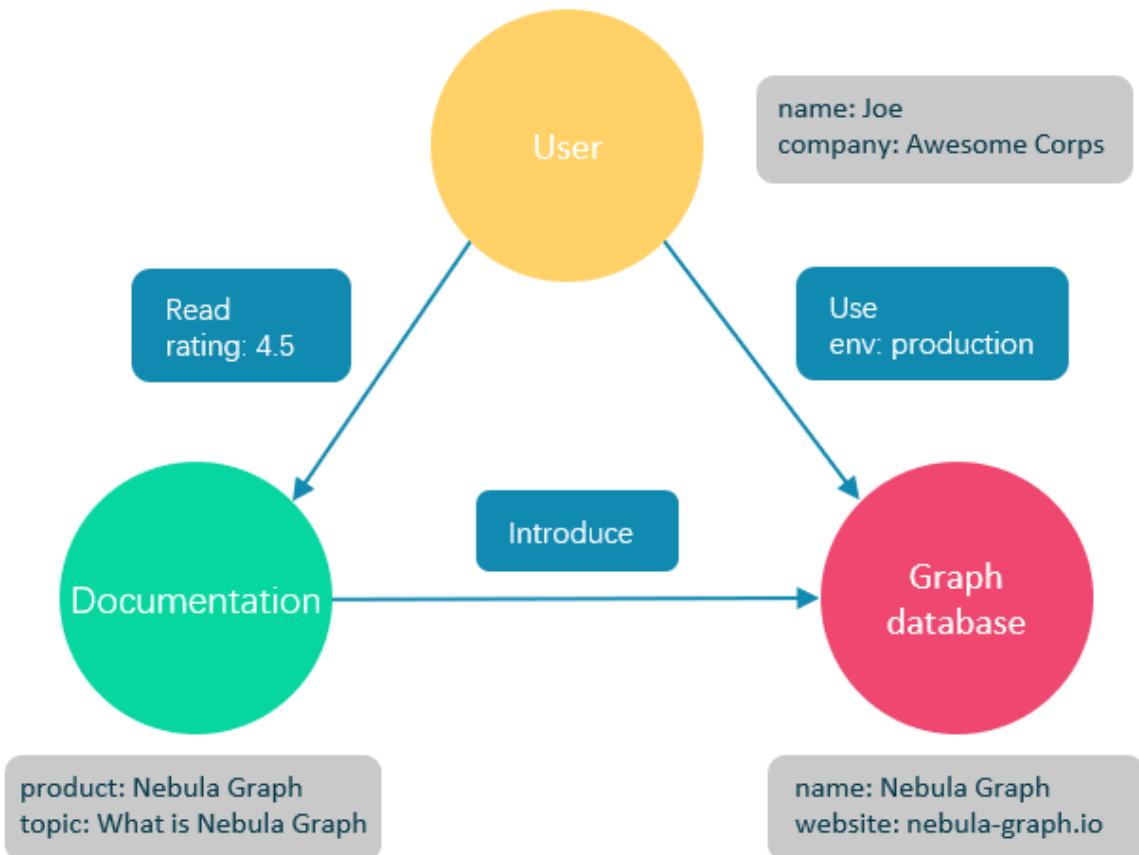
2.1 What is Nebula Graph

Nebula Graph is an open-source, distributed, easily scalable, and native graph database. It is capable of hosting graphs with hundreds of billions of vertices and trillions of edges, and serving queries with millisecond-latency.



2.1.1 What is a graph database

A graph database, such as Nebula Graph, is a database that specializes in storing vast graph networks and retrieving information from them. It efficiently stores data as vertices (nodes) and edges (relationships) in labeled property graphs. Properties can be attached to both vertices and edges. Each vertex can have one or multiple tags (labels).



Graph databases are well suited for storing most kinds of data models abstracted from reality. Things are connected in almost all fields in the world. Modeling systems like relational databases extract the relationships between entities and squeeze them into table columns alone, with their types and properties stored in other columns or even other tables. This makes the data management time-consuming and cost-ineffective.

Nebula Graph, as a typical native graph database, allows you to store the rich relationships as edges with edge types and properties directly attached to them.

2.1.2 Benefits of Nebula Graph

Open-source

Nebula Graph is open under the Apache 2.0 and the Commons Clause 1.0 licenses. More and more people such as database developers, data scientists, security experts, and algorithm engineers are participating in the designing and development of Nebula Graph. To join the opening of source code and ideas, surf the [Nebula Graph GitHub page](#).

Outstanding performance

Written in C++ and born for graph, Nebula Graph handles graph queries in milliseconds. Among most databases, Nebula Graph shows superior performance in providing graph data services. The larger the data size, the greater the superiority of Nebula Graph. For more information, see [Nebula Graph benchmarking](#).

High scalability

Nebula Graph is designed in a shared-nothing architecture and supports scaling in and out without interrupting the database service.

Developer friendly

Nebula Graph supports clients in popular programming languages like Java, Python, C++, and Go, and more are being developed. For more information, see [Nebula Graph clients](#).

Reliable access control

Nebula Graph supports strict role-based access control and external authentication servers such as LDAP (Lightweight Directory Access Protocol) servers to enhance data security. For more information, see [Authentication and authorization](#).

Diversified ecosystem

More and more native tools of Nebula Graph have been released, such as [Nebula Graph Studio](#), [Nebula Console](#), and [Nebula Exchange](#).

Besides, Nebula Graph has the ability to be integrated with many cutting-edge technologies, such as Spark, Flink, and HBase, for the purpose of mutual strengthening in a world of increasing challenges and chances. For more information, see [Ecosystem development](#).

OpenCypher-compatible query language

The native Nebula Graph Query Language, also known as nGQL, is a declarative, openCypher-compatible textual query language. It is easy to understand and easy to use. For more information, see [nGQL guide](#).

Future-oriented hardware with balanced reading and writing

Solid-state drives have extremely high performance and [they are getting cheaper](#). Nebula Graph is a product based on SSD. Compared with products based on HDD and large memory, it is more suitable for future hardware trends and easier to achieve balanced reading and writing.

Easy data modeling and high flexibility

You can easily model the connected data into Nebula Graph for your business without forcing them into a structure such as a relational table, and properties can be added, updated, and deleted freely. For more information, see [Data modeling](#).

High popularity

Nebula Graph is being used by tech leaders such as Tencent, Vivo, Meituan, and JD Digits. For more information, visit the [Nebula Graph official website](#).

2.1.3 Use cases

Nebula Graph can be used to support various graph-based scenarios. To spare the time spent on pushing the kinds of data mentioned in this section into relational databases and on bothering with join queries, use Nebula Graph.

Fraud detection

Financial institutions have to traverse countless transactions to piece together potential crimes and understand how combinations of transactions and devices might be related to a single fraud scheme. This kind of scenario can be modeled in graphs, and with the help of Nebula Graph, fraud rings and other sophisticated scams can be easily detected.

Real-time recommendation

Nebula Graph offers the ability to instantly process the real-time information produced by a visitor and make accurate recommendations on articles, videos, products, and services.

Intelligent question-answer system

Natural languages can be transformed into knowledge graphs and stored in Nebula Graph. A question organized in a natural language can be resolved by a semantic parser in an intelligent question-answer system and re-organized. Then, possible answers to the question can be retrieved from the knowledge graph and provided to the one who asked the question.

Social networking

Information on people and their relationships are typical graph data. Nebula Graph can easily handle the social networking information of billions of people and trillions of relationships, and provide lightning-fast queries for friend recommendations and job promotions in the case of massive concurrency.

2.1.4 Related links

- [Official website](#)
 - [Docs](#)
 - [Blog](#)
 - [Forum](#)
 - [GitHub](#)
-

Last update: August 19, 2021

2.2 Data modeling

A data model is a model that organizes data and specifies how they are related to one another. This topic describes the Nebula Graph data model and provides suggestions for data modeling with Nebula Graph.

2.2.1 Data structures

Nebula Graph data model uses six data structures to store data. They are graph spaces, vertices, edges, tags, edge types and properties.

- **Graph spaces:** Graph spaces are used to isolate data from different teams or programs. Data stored in different graph spaces are securely isolated. Storage replications, privileges, and partitions can be assigned.
- **Vertices:** Vertices are used to store entities.
 - In Nebula Graph, vertices are identified with vertex identifiers (i.e. `VID`). The `VID` must be unique in the same graph space. `VID` should be `int64`, or `fixed_string(N)`.
 - A vertex must have at least one tag or multiple tags.
- **Edges:** Edges are used to connect vertices. An edge is a connection or behavior between two vertices.
 - There can be multiple edges between two vertices.
 - Edges are directed. `->` identifies the directions of edges. Edges can be traversed in either direction.
 - An edge is identified uniquely with a source vertex, an edge type, a rank value, and a destination vertex. Edges have no `EID`.
 - An edge must have one and only one edge type.
 - The rank value is an immutable user-assigned 64-bit signed integer. It identifies the edges with the same edge type between two vertices. Edges are sorted by their rank values. The edge with the greatest rank value is listed first. The default rank value is zero.
- **Tags:** Tags are used to categorize vertices. Vertices that have the same tag share the same definition of properties.
- **Edge types:** Edge types are used to categorize edges. Edges that have the same edge type share the same definition of properties.
- **Properties:** Properties are key-value pairs. Both vertices and edges are containers for properties.

Note

Tag and Edge type are similar to the vertex table and edge table in the relational databases.

2.2.2 Directed property graph

Nebula Graph stores data in directed property graphs. A directed property graph has a set of vertices connected by directed edges. Both vertices and edges can have properties. A directed property graph is represented as:

$$G = \langle V, E, P_V, P_E \rangle$$

- **V** is a set of vertices.
- **E** is a set of directed edges.
- **P_V** is the property of vertices.
- **P_E** is the property of edges.

The following table is an example of the structure of the basketball player dataset. We have two types of vertices, that is **player** and **team**, and two types of edges, that is **serve** and **follow**.

Element	Name	Property name (Data type)	Description
Tag	player	name (string) age (int)	Represents players in the team.
Tag	team	name (string)	Represents the teams.
Edge type	serve	start_year (int) end_year (int)	Represents actions taken by players in the team. An action links a player with a team, and the direction is from a player to a team.
Edge type	follow	degree (int)	Represents actions taken by players in the team. An action links a player with another player, and the direction is from one player to the other player.

Note

Nebula Graph supports only directed edges.

Compatibility

Nebula Graph 2.5.0 allows dangling edges. Therefore, when adding or deleting, you need to ensure the corresponding source vertex and destination vertex of an edge exist. For details, see [INSERT VERTEX](#), [DELETE VERTEX](#), [INSERT EDGE](#), and [DELETE EDGE](#).

The MERGE statement in openCypher is not supported.

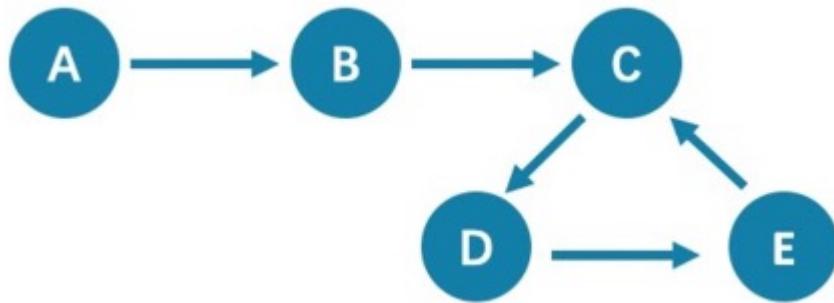
Last update: August 19, 2021

2.3 Path types

In graph theory, a path in a graph is a finite or infinite sequence of edges which joins a sequence of vertices. Paths are fundamental concepts of graph theory.

Paths can be categorized into 3 types: `walk`, `trail`, and `path`. For more information, see [Wikipedia](#).

The following picture is an example for a brief introduction.



2.3.1 walk

A `walk` is a finite or infinite sequence of edges. Both vertices and edges can be repeatedly visited in graph traversal.

In the above picture C, D, and E form a cycle. So, this picture contains infinite paths, such as `A->B->C->D->E`, `A->B->C->D->E->C`, and `A->B->C->D->E->C->D`.

Note

GO statements use `walk`.

2.3.2 trail

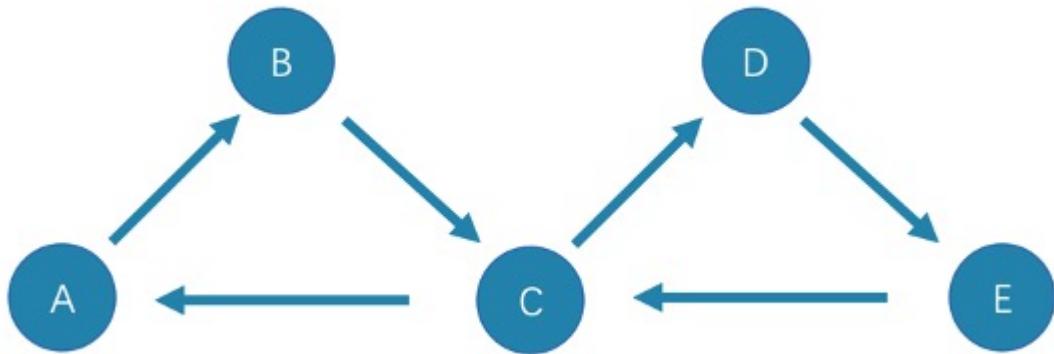
A `trail` is a finite sequence of edges. Only vertices can be repeatedly visited in graph traversal. The Seven Bridges of Königsberg is a typical `trail`.

In the above picture, edges cannot be repeatedly visited. So, this picture contains finite paths. The longest path in this picture consists of 5 edges: `A->B->C->D->E->C`.

Note

MATCH, FIND PATH, and GET SUBGRAPH statements use `trail`.

There are two special cases of trail, `cycle`, and `circuit`. The following picture is an example for a brief introduction.



- cycle

A **cycle** refers to a closed **trail**. Only the terminal vertices can be repeatedly visited. The longest path in this picture consists of 3 edges: `A->B->C->A` or `C->D->E->C`.

- circuit

A **circuit** refers to a closed **trail**. Edges cannot be repeatedly visited in graph traversal. Apart from the terminal vertices, other vertices can also be repeatedly visited. The longest path in this picture: `A->B->C->D->E->C->A`.

2.3.3 path

A **path** is a finite sequence of edges. Neither vertices nor edges can be repeatedly visited in graph traversal.

So, the above picture contains finite paths. The longest path in this picture consists of 4 edges: `A->B->C->D->E`.

.....

Last update: June 25, 2021

2.4 VID

In Nebula Graph, a vertex is uniquely identified by its ID, which is called a VID or a Vertex ID.

2.4.1 Features

- The data types of VIDs are restricted to `FIXED_STRING(<N>)` or `INT64`; a graph space can only select one VID type.
- A VID in a graph space is unique. It functions just as a primary key in a relational database. VIDs in different graph spaces are independent.
- The VID generation method must be set by users, because Nebula Graph does not provide auto increasing ID, or UUID.
- Vertices with the same VID will be identified as the same one. For example:
 - A VID is the unique identifier of an entity, like a person's ID card number. A tag means the type of an entity, such as driver, and boss. Different tags define two groups of different properties, such as driving license number, driving age, order amount, order taking alt, and job number, payroll, debt ceiling, business phone number.
 - When two `INSERT` statements (neither uses a parameter of `IF NOT EXISTS`) with the same VID and tag are operated at the same time, the latter `INSERT` will overwrite the former.
 - When two `INSERT` statements with the same VID but different tags, like `TAG A` and `TAG B`, are operated at the same time, the operation of `Tag A` will not affect `Tag B`.
- VIDs will usually be indexed and stored into memory (in the way of LSM-tree). Thus, direct access to VIDs enjoys peak performance.

2.4.2 VID Operation

- Nebula Graph 1.x only supports `INT64` while Nebula Graph 2.x supports `INT64` and `FIXED_STRING(<N>)`. In `CREATE SPACE`, VID types can be set via `vid_type`.
- `id()` function can be used to specify or locate a VID.
- `LOOKUP` or `MATCH` statements can be used to find a VID via property index.
- Direct access to vertices statements via VIDs enjoys peak performance, such as `DELETE xxx WHERE id(xxx) == "player100"` or `GO FROM "player100"`. Finding VIDs via properties and then operating the graph will cause poor performance, such as `LOOKUP | GO FROM $-.ids`, which will run both `LOOKUP` and `GO` one more time.

2.4.3 VID Generation

VIDs can be generated via applications. Here are some tips:

- (Optimal) Directly take a unique primary key or property as a VID. Property access depends on the VID.
- Generate a VID via a unique combination of properties. Property access depends on property index.
- Generate a VID via algorithms like snowflake. Property access depends on property index.
- If short primary keys greatly outnumber long primary keys, do not enlarge the `N` of `FIXED_STRING(<N>)` too much. Otherwise, it will occupy a lot of memory and hard disks, and slow down performance. Generate VIDs via BASE64, MD5, hash by encoding and splicing.
- If you generate `int64` VID via hash, the probability of collision is about 1/10 when there are 1 billion vertices. The number of edges has no concern with the probability of collision.

2.4.4 Define and modify the data type of VIDs

The data type of VIDs must be defined when you [create the graph space](#). Once defined, it cannot be modified.

Last update: August 19, 2021

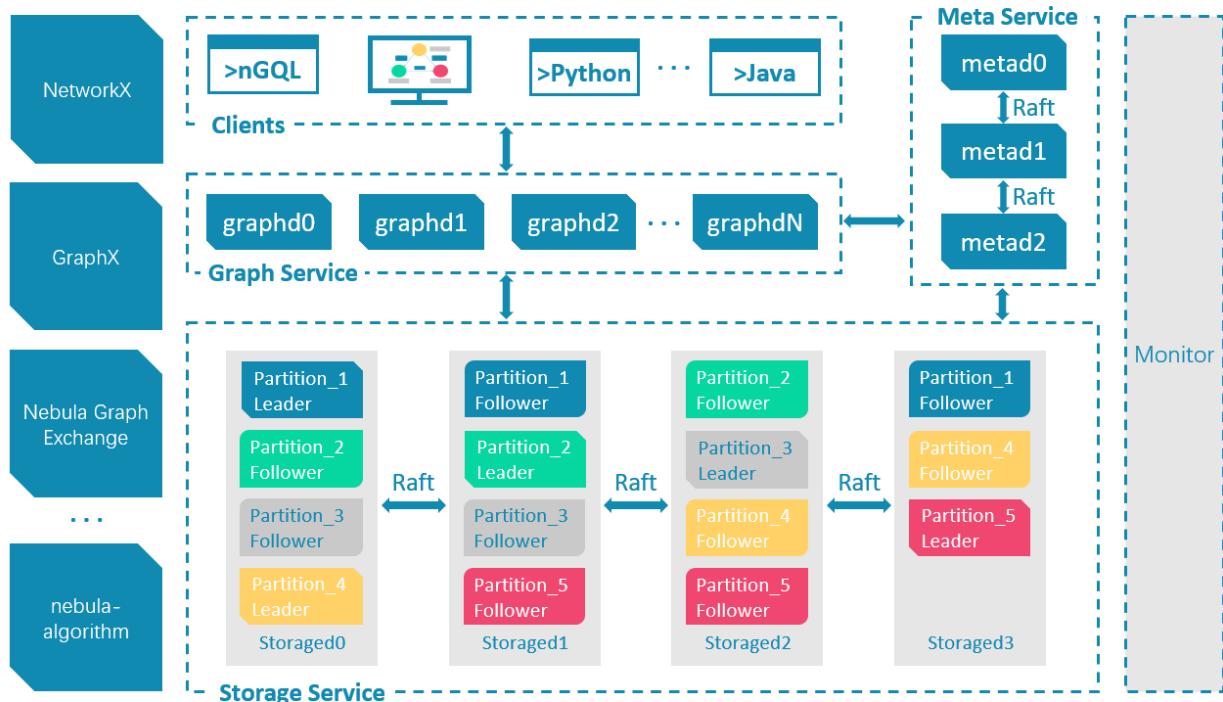
2.5 Nebula Graph architecture

2.5.1 Architecture overview

Nebula Graph consists of three services: the Graph Service, the Storage Service, and the Meta Service. It applies the separation of storage and computing architecture.

Each service has its executable binaries and processes launched from the binaries. Users can deploy a Nebula Graph cluster on a single machine or multiple machines using these binaries.

The following figure shows the architecture of a typical Nebula Graph cluster.



The Meta Service

The Meta Service in the Nebula Graph architecture is run by the `nebula-metad` processes. It is responsible for metadata management, such as schema operations, cluster administration, and user privilege management.

For details on the Meta Service, see [Meta Service](#).

The Graph Service and the Storage Service

Nebula Graph applies the separation of storage and computing architecture. The Graph Service is responsible for querying. The Storage Service is responsible for storage. They are run by different processes, i.e., nebula-graphd and nebula-storaged. The benefits of the separation of storage and computing architecture are as follows:

- Great scalability

The separated structure makes both the Graph Service and the Storage Service flexible and easy to scale in or out.

- High availability

If part of the Graph Service fails, the data stored by the Storage Service suffers no loss. And if the rest part of the Graph Service is still able to serve the clients, service recovery can be performed quickly, even unfelt by the users.

- Cost-effective

The separation of storage and computing architecture provides a higher resource utilization rate, and it enables clients to manage the cost flexibly according to business demands. The cost savings can be more distinct if the [Nebula Graph Cloud](#) service is used.

- Open to more possibilities

With the ability to run separately, the Graph Service may work with multiple types of storage engines, and the Storage Service may also serve more types of computing engines.

For details on the Graph Service and the Storage Service, see [Graph Service](#) and [Storage Service](#).

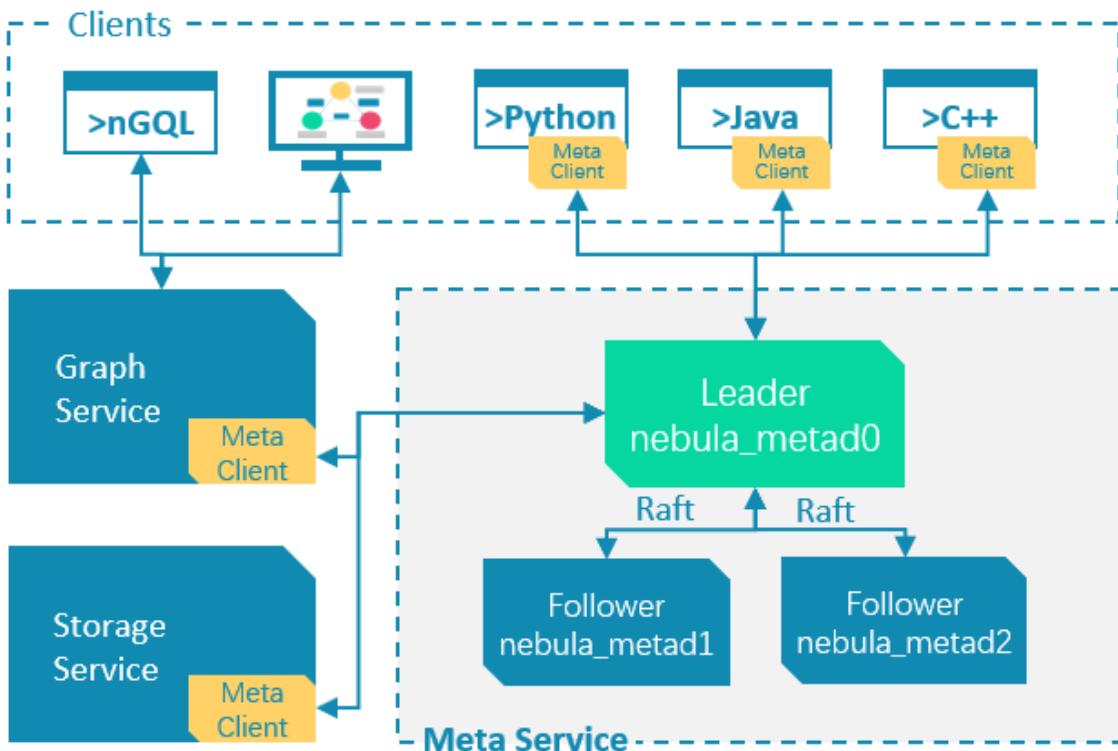
Last update: July 1, 2021

2.5.2 Meta Service

This topic introduces the architecture and functions of the Meta Service.

The architecture of the Meta Service

The architecture of the Meta Service is as follows:



The Meta Service is run by nebula-metad processes. Users can deploy nebula-metad processes according to the scenario:

- In a test environment, users can deploy one or three nebula-metad processes on different machines or a single machine.
- In a production environment, we recommend that users deploy three nebula-metad processes on different machines for high availability.

All the nebula-metad processes form a Raft-based cluster, with one process as the leader and the others as the followers.

The leader is elected by the majorities and only the leader can provide service to the clients or other components of Nebula Graph. The followers will be run in a standby way and each has a data replication of the leader. Once the leader fails, one of the followers will be elected as the new leader.

Note

The data of the leader and the followers will keep consistent through Raft. Thus the breakdown and election of the leader will not cause data inconsistency. For more information on Raft, see [Storage service architecture](#).

Functions of the Meta Service

MANAGES USER ACCOUNTS

The Meta Service stores the information of user accounts and the privileges granted to the accounts. When the clients send queries to the Meta Service through an account, the Meta Service checks the account information and whether the account has the right privileges to execute the queries or not.

For more information on Nebula Graph access control, see [Authentication and authorization](#).

MANAGES PARTITIONS

The Meta Service stores and manages the locations of the storage partitions and helps balance the partitions.

MANAGES GRAPH SPACES

Nebula Graph supports multiple graph spaces. Data stored in different graph spaces are securely isolated. The Meta Service stores the metadata of all graph spaces and tracks the changes of them, such as adding or dropping a graph space.

MANAGES SCHEMA INFORMATION

Nebula Graph is a strong-typed graph database. Its schema contains tags (i.e., the vertex types), edge types, tag properties, and edge type properties.

The Meta Service stores the schema information. Besides, it performs the addition, modification, and deletion of the schema, and logs the versions of them.

For more information on Nebula Graph schema, see [Data model](#).

MANAGES TTL-BASED DATA EVICTION

The Meta Service provides automatic data eviction and space reclamation based on TTL (time to live) options for Nebula Graph.

For more information on TTL, see [TTL options](#).

MANAGES JOBS

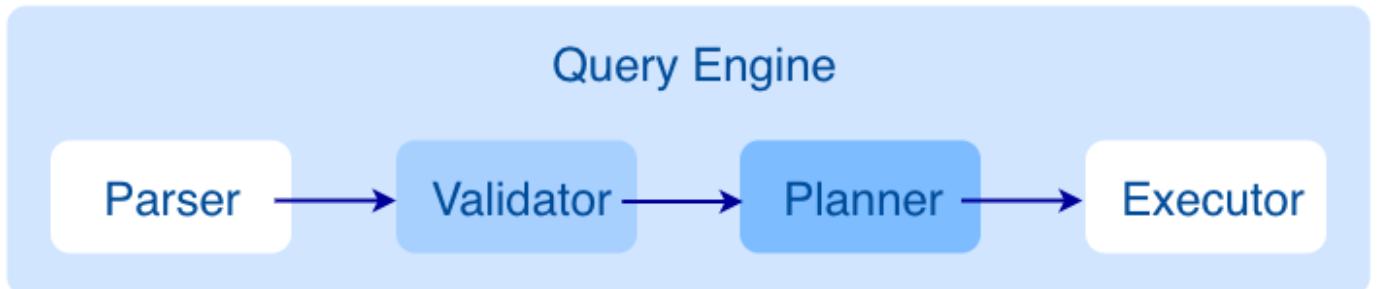
The Job Management module in the Meta Service is responsible for the creation, queuing, querying, and deletion of jobs.

Last update: August 24, 2021

2.5.3 Graph Service

Graph Service is used to process the query. It has four submodules: Parser, Validator, Planner, and Executor. This topic will describe Graph Service accordingly.

The architecture of Graph Service



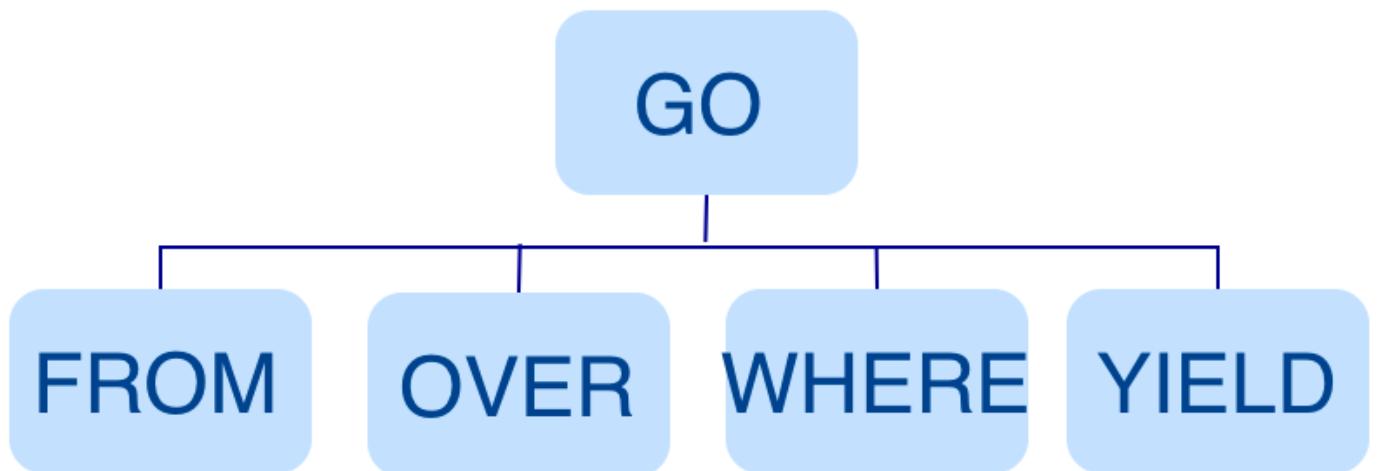
After a query is sent to Graph Service, it will be processed by the following four submodules:

1. **Parser** Performs lexical analysis and syntax analysis.
2. **Validator** Validates the statements.
3. **Planner** Generates and optimizes the execution plans.
4. **Executor** Executes the operators.

Parser

After receiving a request, the statements will be parsed by the Parser composed of Flex (lexical analysis tool) and Bison (syntax analysis tool), and its corresponding AST will be generated. Statements will be directly intercepted in this stage because of its invalid syntax.

For example, the structure of the AST of `GO FROM "Tim" OVER like WHERE like.likeness > 8.0 YIELD like._dst` is shown in the following picture.



Validator

Validator performs a series of validations on the AST. It mainly works on these tasks:

- Validating metadata

Validator will validate whether the metadata is correct or not.

When parsing the `OVER`, `WHERE`, and `YIELD` clauses, Validator looks up the Schema and verifies whether the edge type and tag data exist or not. For an `INSERT` statement, Validator verifies whether the types of the inserted data are the same as the ones defined in the Schema.

- Validating contextual reference

Validator will verify whether the cited variable exists or not, or whether the cited property is variable or not.

For composite statements, like `$var = GO FROM "Tim" OVER like YIELD like._dst AS ID; GO FROM $var.ID OVER serve YIELD serve._dst`, Validator verifies first to see if `var` is defined, and then to check if the `ID` property is attached to the `var` variable.

- Validating type inference

Validator infers what type the result of an expression is and verifies the type against the specified clause.

For example, the `WHERE` clause requires the result to be a `bool` value, a `NULL` value, or `empty`.

- Validating the information of `*`

Validator needs to verify all the Schema that involves `*` when verifying the clause if there is a `*` in the statement.

Take a statement like `GO FROM "Tim" OVER * YIELD like._dst, like.likeness, serve._dst` as an example. When verifying the `OVER` clause, Validator needs to verify all the edge types. If the edge type includes `like` and `serve`, the statement would be `GO FROM "Tim" OVER like,serve YIELD like._dst, like.likeness, serve._dst`.

- Validating input and output

Validator will check the consistency of the clauses before and after the `|`.

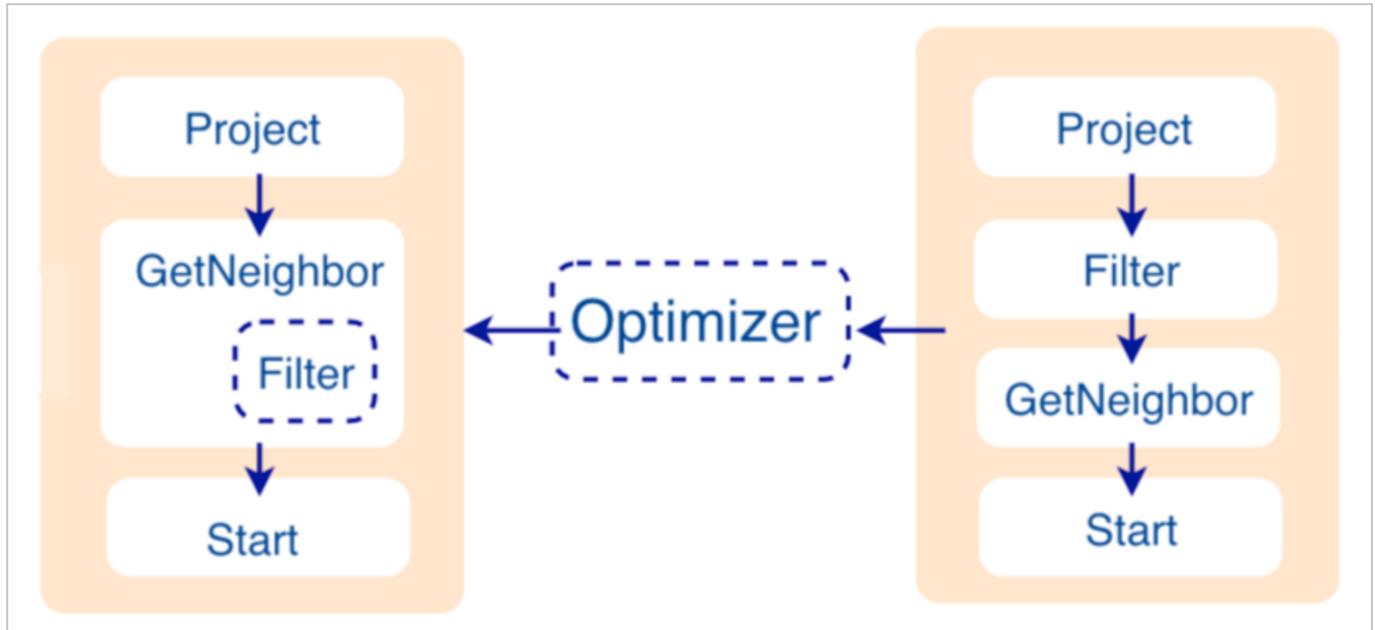
In the statement `GO FROM "Tim" OVER like YIELD like._dst AS ID | GO FROM $-.ID OVER serve YIELD serve._dst`, Validator will verify whether `$-.ID` is defined in the clause before the `|`.

When the validation succeeds, an execution plan will be generated. Its data structure will be stored in the `src/planner` directory.

Planner

In the `nebula-graphd.conf` file, when `enable_optimizer` is set to be `false`, Planner will not optimize the execution plans generated by Validator. It will be executed by Executor directly.

In the `nebula-graphd.conf` file, when `enable_optimizer` is set to be `true`, Planner will optimize the execution plans generated by Validator. The structure is as follows.



- Before optimization

In the execution plan on the right side of the preceding picture, each node directly depends on other nodes. For example, the root node `Project` depends on the `Filter` node, the `Filter` node depends on the `GetNeighbor` node, and so on, up to the leaf node `Start`. Then the execution plan is (not truly) executed.

During this stage, every node has its input and output variables, which are stored in a hash table. The execution plan is not truly executed, so the value of each key in the associated hash table is empty (except for the `Start` node, where the input variables hold the starting data), and the hash table is defined in `src/context/ExecutionContext.cpp` under the `nebula-graph` repository.

For example, if the hash table is named as `ResultMap` when creating the `Filter` node, users can determine that the node takes data from `ResultMap["GN1"]`, then puts the result into `ResultMap["Filter2"]`, and so on. All these work as the input and output of each node.

- Process of optimization

The optimization rules that Planner has implemented so far are considered RBO (Rule-Based Optimization), namely the pre-defined optimization rules. The CBO (Cost-Based Optimization) feature is under development. The optimized code is in the `src/optimizer/` directory under the `nebula-graph` repository.

RBO is a “bottom-up” exploration process. For each rule, the root node of the execution plan (in this case, the `Project` node) is the entry point, and step by step along with the node dependencies, it reaches the node at the bottom to see if it matches the rule.

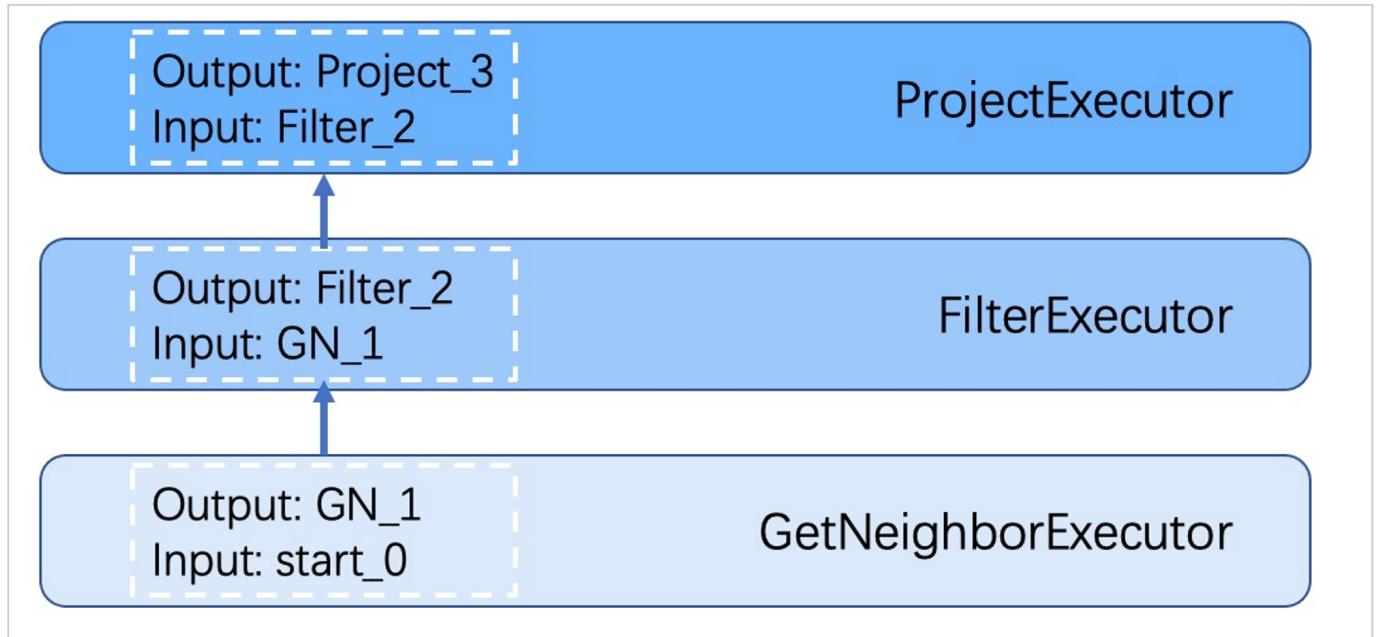
As shown in the preceding figure, when the `Filter` node is explored, it is found that its children node is `GetNeighbors`, which matches successfully with the pre-defined rules, so a transformation is initiated to integrate the `Filter` node into the `GetNeighbors` node, the `Filter` node is removed, and then the process continues to the next rule. Therefore, when the `GetNeighbor` operator calls interfaces of the Storage layer to get the neighboring edges of a vertex during the execution stage, the Storage layer will directly filter out the unqualified edges internally. Such optimization greatly reduces the amount of data transfer, which is commonly known as filter pushdown.

 **Note**

Nebula Graph 2.5.0 will not run optimization by default.

Executor

The Executor module consists of Scheduler and Executor. The Scheduler generates the corresponding execution operators against the execution plan, starting from the leaf nodes and ending at the root node. The structure is as follows.



Each node of the execution plan has one execution operator node, whose input and output have been determined in the execution plan. Each operator only needs to get the values for the input variables, compute them, and finally put the results into the corresponding output variables. Therefore, it is only necessary to execute step by step from `start`, and the result of the last operator is returned to the user as the final result.

Source code hierarchy

The source code hierarchy under the nebula-graph repository is as follows.

```

|--src
  |--context  //contexts for validation and execution
  |--daemons
  |--executor //execution operators
  |--mock
  |--optimizer //optimization rules
  |--parser   //lexical analysis and syntax analysis
  |--planner  //structure of the execution plans
  |--scheduler //scheduler
  |--service
  |--util     //basic components
  |--validator //validation of the statements
  |--visitor
  
```

Last update: August 27, 2021

2.5.4 Storage Service

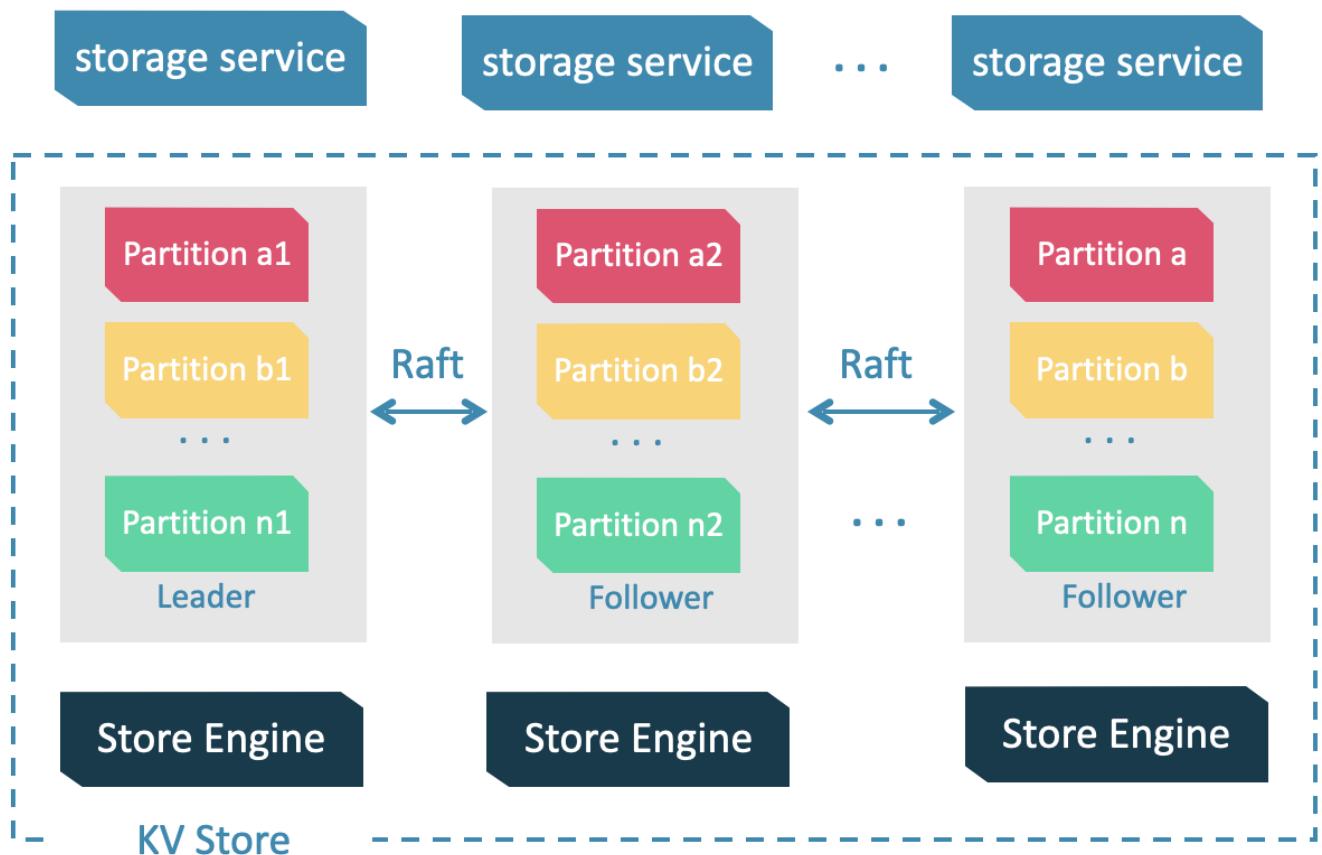
The persistent data of Nebula Graph have two parts. One is the [Meta Service](#) that stores the meta-related data.

The other is the Storage Service that stores the data, which is run by the nebula-storaged process. This topic will describe the architecture of Storage Service.

Advantages

- High performance (Customized built-in KVStore)
- Great scalability (Shared-nothing architecture, not rely on NAS/SAN-like devices)
- Strong consistency (Raft)
- High availability (Raft)
- Supports synchronizing with the third party systems, such as [Elasticsearch](#).

The architecture of Storage Service



Storage Service is run by the nebula-storaged process. Users can deploy nebula-storaged processes on different occasions. For example, users can deploy 1 nebula-storaged process in a test environment and deploy 3 nebula-storaged processes in a production environment.

All the nebula-storaged processes consist of a Raft-based cluster. There are three layers in the Storage Service:

- Storage interface

The top layer is the storage interface. It defines a set of APIs that are related to the graph concepts. These API requests will be translated into a set of KV operations targeting the corresponding [Partition](#). For example:

- `getNeighbors` : query the in-edge or out-edge of a set of vertices, return the edges and the corresponding properties, and support conditional filtering.
- `insert vertex/edge` : insert a vertex or edge and its properties.
- `getProps` : get the properties of a vertex or an edge.

It is this layer that makes the Storage Service a real graph storage. Otherwise, it is just a KV storage.

- Consensus

Below the storage interface is the consensus layer that implements [Multi Group Raft](#), which ensures the strong consistency and high availability of the Storage Service.

- Store engine

The bottom layer is the local storage engine library, providing operations like `get`, `put`, and `scan` on local disks. The related interfaces are stored in `kvstore.h` and `KVEngine.h` files. Users can develop their own local store plugins based on their needs.

The following will describe some features of Storage Service based on the above architecture.

KVStore

Nebula Graph develops and customizes its built-in KVStore for the following reasons.

- It is a high-performance KVStore.
- It is provided as a (kv) library and can be easily developed for the filtering-pushdown purpose. As a strong-typed database, how to provide Schema during pushdown is the key to efficiency for Nebula Graph.
- It has strong data consistency.

Therefore, Nebula Graph develops its own KVStore with RocksDB as the local storage engine. The advantages are as follows.

- For multiple local hard disks, Nebula Graph can make full use of its concurrent capacities through deploying multiple data directories.
- Meta Service manages all the Storage servers. All the partition distribution data and current machine status can be found in the meta service. Accordingly, users can execute a manual load balancing plan in meta service.

 **Note**

Nebula Graph does not support auto load balancing because auto data transfer will affect online business.

- Nebula Graph provides its own WAL mode so one can customize the WAL. Each partition owns its WAL.
- One Nebula Graph KVStore cluster supports multiple graph spaces, and each graph space has its own partition number and replica copies. Different graph spaces are isolated physically from each other in the same cluster.

Data storage formats

Nebula Graph stores vertices and edges. Efficient property filtering is critical for a Graph Database. So, Nebula Graph uses keys to store vertices and edges, while uses values to store the related properties.

Nebula Graph 2.0 has changed a lot over its releases. The following will introduce the old and new data storage formats and cover their differences.

- Vertex format

Vertex					
V1.x	Type (1 byte)	PartID (3 bytes)	VertexID (8 bytes)	TagID (4 bytes)	Timestamp (8 bytes)
V2.x	Type (1 byte)	PartID (3 bytes)	VertexID (n bytes)	TagID (4 bytes)	

Field	Description
Type	One byte, used to indicate the key type.
PartID	Three bytes, used to indicate the sharding partition and to scan the partition data based on the prefix when re-balancing the partition.
VertexID	Used to indicate vertex ID. For an integer VertexID, it occupies eight bytes. However, for a string VertexID, it is changed to <code>fixed_string</code> of a fixed length which needs to be specified by users when they create the space.
TagID	Four bytes, used to indicate the tags that vertex relate with.

- Edge Format

Edge						
V1.x	Type (1 byte)	PartID (3 bytes)	VertexID (8 bytes)	Edge Type (4 bytes)	Rank (8 bytes)	VertexID (8 bytes)
V2.x	Type (1 byte)	PartID (3 bytes)	VertexID (n bytes)	Edge Type (4 bytes)	Rank (8 bytes)	VertexID (n bytes) PlaceHolder (1 byte)

Field	Description
Type	One byte, used to indicate the key type.
PartID	Three bytes, used to indicate the sharding partition. This field can be used to scan the partition data based on the prefix when re-balancing the partition.
VertexID	Used to indicate vertex ID. The former VID refers to source VID in out-edge and dest VID in in-edge, while the latter VID refers to dest VID in out-edge and source VID in in-edge.
Edge Type	Four bytes, used to indicate edge type. Greater than zero means out-edge, less than zero means in-edge.
Rank	Eight bytes, used to indicate multiple edges in one edge type. Users can set the field based on needs and store weight, such as transaction time and transaction number.
PlaceHolder	One byte, used to indicate a placeholder, which is purposely designed for TOSS (Transaction On Storage Side).

Legacy version compatibility

The differences between Nebula Graph 1.x and 2.0 are as follows:

- In Nebula Graph 1.x, a vertex and an edge have the same `Type` byte, while in Nebula Graph 2.0, the `Type` byte differs from each other, which separates vertices and edges physically so that all tags of a vertex can be easily queried.
- Nebula Graph 1.x supports only int IDs, while Nebula Graph 2.0 is compatible with both int IDs and string IDs.
- Nebula Graph 2.0 removes `Timestamp` in both vertex and edge key formats.
- Nebula Graph 2.0 adds `PlaceHolder` to edge key format.
- Nebula Graph 2.0 has changed the formats of indexes for a range query.

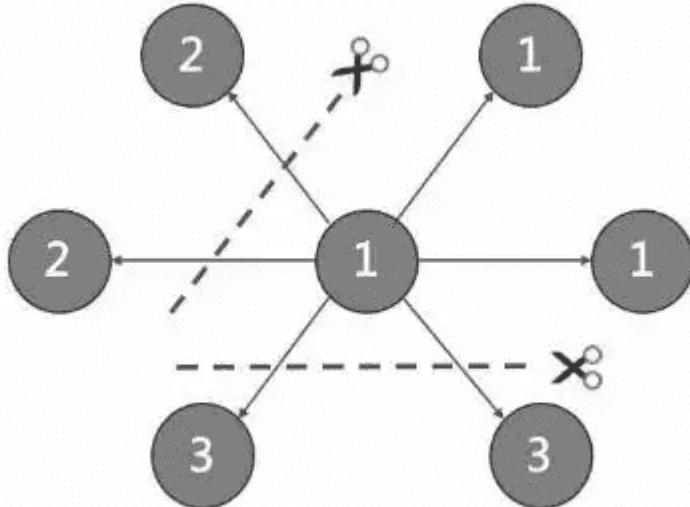
PROPERTY DESCRIPTIONS

Nebula Graph uses strong-typed Schema.

Nebula Graph will store the properties of vertex and edges in order after encoding them. Since the length of properties is fixed, queries can be made in no time according to offset. Before decoding, Nebula Graph needs to get (and cache) the schema information in the Meta Service. In addition, when encoding properties, Nebula Graph will add the corresponding schema version to support online schema change.

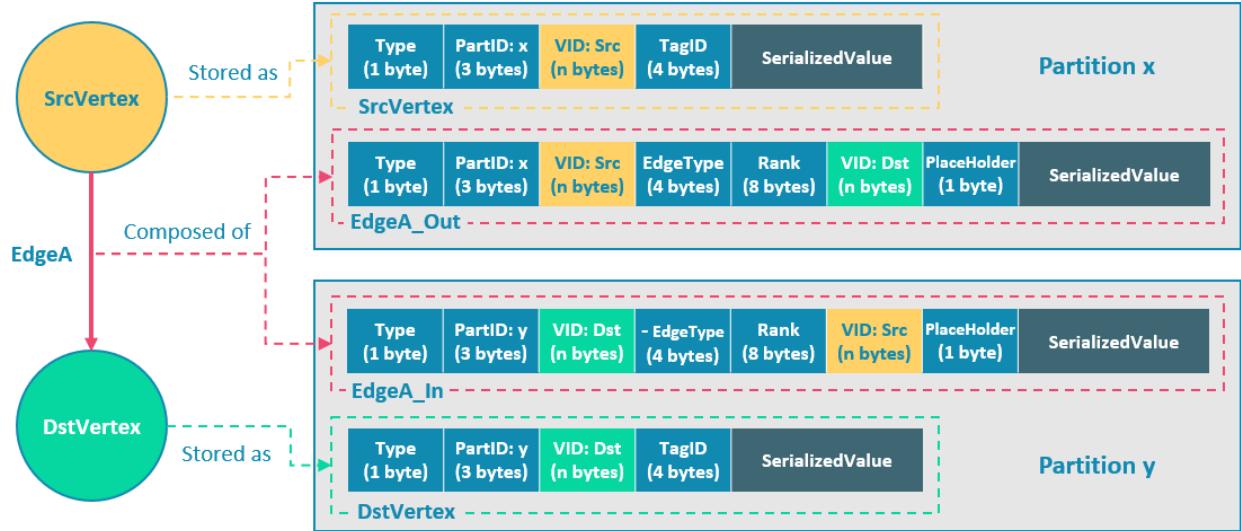
Data partitioning

Since in an ultra-large-scale relational network, vertices can be as many as tens to hundreds of billions, and edges are even more than trillions. Even if only vertices and edges are stored, the storage capacity of both exceeds that of ordinary servers. Therefore, Nebula Graph uses hash to shard the graph elements and store them in different partitions.



EDGE AND STORAGE AMPLIFICATION

In Nebula Graph, an edge corresponds to two key-value pairs on the hard disk. When there are lots of edges and each has many properties, storage amplification will be obvious. The storage format of edges is shown in the picture below.



In this example, **SrcVertex** connects **DstVertex** via **EdgeA**, forming the path of `(SrcVertex)-[EdgeA]->(DstVertex)`. **SrcVertex**, **DstVertex**, and **EdgeA** will all be stored in **Partition x** and **Partition y** as four key-value pairs in the storage layer. Details are as follows:

- The key value of **SrcVertex** is stored in **Partition x**. Key fields include Type, PartID(x), VID(Src), and TagID. SerializedValue, namely Value, refers to serialized vertex properties.
- The first key value of **EdgeA**, namely **EdgeA_Out**, is stored in the same partition as the **SrcVertex**. Key fields include Type, PartID(x), VID(Src), EdgeType(+ means out-edge), Rank(0), VID(Dst), and PlaceHolder. SerializedValue, namely Value, refers to serialized edge properties.
- The key value of **DstVertex** is stored in **Partition y**. Key fields include Type, PartID(y), VID(Dst), and TagID. SerializedValue, namely Value, refers to serialized vertex properties.
- The second key value of **EdgeA**, namely **EdgeA_In**, is stored in the same partition as the **DstVertex**. Key fields include Type, PartID(y), VID(Dst), EdgeType(- means in-edge), Rank(0), VID(Src), and PlaceHolder. SerializedValue, namely Value, refers to serialized edge properties, which is exactly the same as that in **EdgeA_Out**.

EdgeA_Out and **EdgeA_In** are stored in storage layer with opposite directions, constituting **EdgeA** logically. **EdgeA_Out** is used for traversal requests starting from **SrcVertex**, such as `(a)-[]->()`; **EdgeA_In** is used for traversal requests starting from **DstVertex**, such as `()-[]->(a)`.

Like **EdgeA_Out** and **EdgeA_In**, Nebula Graph redundantly stores the information of each edge, which doubles the actual capacities needed for edge storage. The key corresponding to the edge occupies a small hard disk space, but the space occupied by Value is proportional to the length and amount of the property value. Therefore, it will occupy a relatively large hard disk space if the property value of the edge is large or there are many edge property values.

PARTITION ALGORITHM

Nebula Graph uses a **static Hash** strategy to shard data through a modulo operation on vertex ID. All the out-keys, in-keys, and tag data will be placed in the same partition. In this way, query efficiency is increased dramatically.

Note

The number of partitions needs to be determined when users are creating a graph space since it cannot be changed afterward. Users are supposed to take into consideration the demands of future business when setting it.

When inserting into Nebula Graph, vertices and edges are distributed across different partitions. And the partitions are located on different machines. The number of partitions is set in the `CREATE SPACE` statement and cannot be changed afterward.

If certain vertices need to be placed on the same partition (i.e., on the same machine), see [Formula/code](#)

The following code will briefly describe the relationship between VID and partition.

```
// If VertexID occupies 8 bytes, it will be stored in int64 to be compatible with the version 1.0.
uint64_t vid = 0;
if (id.size() == 8) {
    memcpy(static_cast<void*>(&vid), id.data(), 8);
} else {
    MurmurHash2 hash;
    vid = hash(id.data());
}
PartitionID pId = vid % numParts + 1;
```

Roughly speaking, after hashing a fixed string to int64, (the hashing of int64 is the number itself), do modulo, and then plus one, namely:

```
pId = vid % numParts + 1;
```

Parameters and descriptions of the preceding formula are as follows:

Parameter	Description
%	The modulo operation.
numParts	The number of partitions for the graph space where the <code>vid</code> is located, namely the value of <code>partition_num</code> in the CREATE SPACE statement.
<code>pId</code>	The ID for the partition where the <code>vid</code> is located.

Suppose there are 100 partitions, the vertices with `vid` 1, 101, and 1001 will be stored on the same partition. But, the mapping between the partition ID and the machine address is random. Therefore, we cannot assume that any two partitions are located on the same machine.

Raft

RAFT IMPLEMENTATION

In a distributed system, one data usually has multiple replicas so that the system can still run normally even if a few copies fail. It requires certain technical means to ensure consistency between replicas.

Basic principle: Raft is designed to ensure consistency between replicas. Raft uses election between replicas, and the (candidate) replica that wins more than half of the votes will become the Leader, providing external services on behalf of all replicas. The rest Followers will play backups. When the Leader fails (due to communication failure, operation and maintenance commands, etc.), the rest Followers will conduct a new round of elections and vote for a new Leader. The Leader and Followers will detect each other's survival through heartbeats and write them to the hard disk in Raft-wal mode. Replicas that do not respond to more than multiple heartbeats will be considered faulty.

Note

Raft-wal needs to be written into the hard disk periodically. If hard disk bottlenecks to write, Raft will fail to send a heartbeat and conduct a new round of elections. If the hard disk IO is severely blocked, there will be no Leader for a long time.

Read and write: For every writing request of the clients, the Leader will initiate a Raft-wal and synchronize it with the Followers. Only after over half replicas have received the Raft-wal will it return to the clients successfully. For every reading request of the clients, it will get to the Leader directly, while Followers will not be involved.

Failure: Scenario 1: Take a (space) cluster of a single replica as an example. If the system has only one replica, the Leader will be itself. If failure happens, the system will be completely unavailable. Scenario 2: Take a (space) cluster of three replicas as an example. If the system has three replicas, one of them will be the Leader and the rest will be the Followers. If the Leader fails, the rest two can still vote for a new Leader (and a Follower), and the system is still available. But if any of the two Followers fails again, the system will be completely unavailable due to inadequate voters.

Note

Raft and HDFS have different modes of duplication. Raft is based on a quorum vote, so the number of replicas cannot be even.

Listener: As is a special role in Raft, it cannot vote or keep data consistency. In Nebula Graph, it reads Raft-wal from the Leader and synchronizes it to ElasticSearch cluster.

MULTI GROUP RAFT

Storage Service supports a distributed cluster architecture, so Nebula Graph implements Multi Group Raft according to Raft protocol. Each Raft group stores all the replicas of each partition. One replica is the leader, while others are followers. In this way, Nebula Graph achieves strong consistency and high availability. The functions of Raft are as follows.

Nebula Graph uses Multi Group Raft to improve performance when there are many partitions because Raft-wal cannot be NULL. When there are too many partitions, costs will increase, such as storing information in Raft group, WAL files, or batch operation in low load.

There are two key points to implement the Multi Raft Group:

- To share transport layer

Each Raft Group sends messages to its corresponding peers. So if the transport layer cannot be shared, the connection costs will be very high.

- To share thread pool

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 a batch to improve throughput when writing the WAL serially. As Nebula Graph uses WAL to implement some special functions, batches need to be grouped, which is a feature of Nebula Graph.

For example, lock-free CAS operations will execute after all the previous WALs are committed. So for a batch, if there are several WALs in CAS type, we need to divide this batch into several smaller groups and make sure they are committed serially.

LISTENER

The Listener is designed for **storage horizontal scaling**. It takes a long time for the newly added machines to be synchronized with data. Therefore, these machines cannot join the group followers, otherwise, the availability of the entire cluster will decrease.

The Listener will write into the command WAL. If the leader finds a command of `add learner` when writing the WAL, it will add the listener to its peers and mark it as a Listener. Listeners cannot join the quorum votes, but logs will still be sent to them as usual. Listeners themselves will not initiate elections.

Raft listener can write the data into Elasticsearch cluster after receiving them from Learner to implement full-text search. For more information, see [Deploy Raft Listener](#) 

TRANSFER LEADERSHIP

Transfer leadership is extremely important for balance. When moving a partition from one machine to another, Nebula Graph first checks if the source is a leader. If so, it should be moved to another peer. After data migration is completed, it is important to [balance leader distribution](#) again.

When a transfer leadership command is committed, the leader will abandon its leadership and the followers will start a leader election.

PEER CHANGES

To avoid split-brain, when members in a Raft Group change, an intermediate state is required. In such a state, the quorum of the old group and new group always have an overlap. Thus it prevents the old or new group from making decisions unilaterally. To make it even simpler, in his doctoral thesis Diego Ongaro suggests adding or removing a peer once to ensure the overlap between the quorum of the new group and the old group. Nebula Graph also uses this approach, except that the way to add or remove a member is different. For details, please refer to `addPeer`/`removePeer` in the Raft Part class.

Differences with HDFS

Storage Service is a Raft-based distributed architecture, which has certain differences with that of HDFS. For example:

- Storage Service ensures consistency through Raft. Usually, the number of its replicas is odd to elect a leader. However, DataNode used by HDFS ensures consistency through NameNode, which has no limit on the number of replicas.
- In Storage Service, only the replicas of the leader can read and write, while in HDFS all the replicas can do so.
- In Storage Service, the number of replicas needs to be determined when creating a space, since it cannot be changed afterward. But in HDFS, the number of replicas can be changed freely.
- Storage Service can access the file system directly. While the applications of HDFS (such as HBase) have to access HDFS before the file system, which requires more RPC times.

In a word, Storage Service is more lightweight with some functions simplified and its architecture is simpler than HDFS, which can effectively improve the read and write performance of a smaller block of data.

Last update: August 27, 2021

3. Quick start

3.1 Quick start workflow

The quick start introduces the simplest workflow to use Nebula Graph, including deploying Nebula Graph, connecting to Nebula Graph, and doing basic CRUD.

3.1.1 Documents

Users can quickly deploy and use Nebula Graph in the following steps.

1. Deploy Nebula Graph

Users can use the RPM or DEB file to quickly deploy Nebula Graph. For other ways to deploy Nebula Graph and corresponding preparations, see [deployment and installation](#).

2. Start Nebula Graph

Users need to start Nebula Graph after deployment.

3. Connect to Nebula Graph

Then users can use clients to connect to Nebula Graph. Nebula Graph supports a variety of clients. This topic will describe how to use Nebula Console to connect to Nebula Graph.

4. CRUD in Nebula Graph

Users can use nGQL (Nebula Graph Query Language) to run CRUD after connecting to Nebula Graph.

Last update: August 23, 2021

3.2 Manage Nebula Graph services

The `nebula.service` script can start, stop, restart, terminate, and check the Nebula Graph services.

`nebula.service` is stored in the `/usr/local/nebula/scripts` directory by default, which is also the default installation path of Nebula Graph. If you have customized the path, use the actual path in your environment.

3.2.1 Syntax

```
$ sudo /usr/local/nebula/scripts/nebula.service
[-v] [-c <config_file_path>]
<start|stop|restart|kill|status>
<metad|graphd|storaged|all>
```

Parameter	Description
<code>-v</code>	Display detailed debugging information.
<code>-c</code>	Specify the configuration file path. The default path is <code>/usr/local/nebula/etc/</code> .
<code>start</code>	Start the target services.
<code>stop</code>	Stop the target services.
<code>restart</code>	Restart the target services.
<code>kill</code>	Terminate the target services.
<code>status</code>	Check the status of the target services.
<code>metad</code>	Set the Meta Service as the target service.
<code>graphd</code>	Set the Graph Service as the target service.
<code>storaged</code>	Set the Storage Service as the target service.
<code>all</code>	Set all the Nebula Graph services as the target services.

3.2.2 Start Nebula Graph services

In non-container environment

About non-container deployment, see [Install Nebula Graph with RPM or DEB package](#). Run the following command to start Nebula Graph.

```
$ 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
```

In docker container (deployed with docker-compose)

Run the following command in the `nebula-docker-compose/` directory to start Nebula Graph services.

```
[nebula-docker-compose]$ docker-compose up -d
Building with native build. Learn about native build in Compose here: https://docs.docker.com/go/compose-native-build/
Creating network "nebula-docker-compose_nebula-net" with the default driver
Creating nebula-docker-compose_metad0_1 ... done
Creating nebula-docker-compose_metad2_1 ... done
Creating nebula-docker-compose_metad1_1 ... done
Creating nebula-docker-compose_storaged2_1 ... done
Creating nebula-docker-compose_graphd1_1 ... done
Creating nebula-docker-compose_storaged1_1 ... done
Creating nebula-docker-compose_storaged0_1 ... done
Creating nebula-docker-compose_graphd2_1 ... done
Creating nebula-docker-compose_graphd1_1 ... done
```

3.2.3 Stop Nebula Graph services

Danger

Do not run `kill -9` to forcibly terminate the processes, otherwise, there will be a low probability of data loss.

In non-container environment

Run the following command to stop Nebula Graph services.

```
$ 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
```

In docker container (deployed with docker-compose)

Run the following command in the `nebula-docker-compose/` directory to stop Nebula Graph services.

```
nebula-docker-compose$ docker-compose down
Stopping nebula-docker-compose_graphd_1 ... done
Stopping nebula-docker-compose_graphd2_1 ... done
Stopping nebula-docker-compose_storaged0_1 ... done
Stopping nebula-docker-compose_storaged1_1 ... done
Stopping nebula-docker-compose_graphd1_1 ... done
Stopping nebula-docker-compose_storaged2_1 ... done
Stopping nebula-docker-compose_metad1_1 ... done
Stopping nebula-docker-compose_metad2_1 ... done
Stopping nebula-docker-compose_metad0_1 ... done
Removing nebula-docker-compose_graphd_1 ... done
Removing nebula-docker-compose_graphd2_1 ... done
Removing nebula-docker-compose_storaged0_1 ... done
Removing nebula-docker-compose_storaged1_1 ... done
Removing nebula-docker-compose_graphd1_1 ... done
Removing nebula-docker-compose_storaged2_1 ... done
Removing nebula-docker-compose_metad1_1 ... done
Removing nebula-docker-compose_metad2_1 ... done
Removing nebula-docker-compose_metad0_1 ... done
Removing network nebula-docker-compose_nebula-net
```

Note

If you are using a developing or nightly version for testing and having compatibility issues, try to run `docker-compose down-v` to **DELETE** all data stored in Nebula Graph and import data again.

3.2.4 Check the service status

In non-container environment

Run the following command to check the service status of Nebula Graph.

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

- Nebula Graph is running normally if the following information is returned.

```
[INFO] nebula-metad: Running as 26601, Listening on 9559
[INFO] nebula-graphd: Running as 26644, Listening on 9669
[INFO] nebula-storaged: Running as 26709, Listening on 9779
```

- If the returned information is similar to the following one, there must be a problem. Users can check the corresponding information to troubleshoot problems or go to [Nebula Graph Forum](#) for help.

```
[INFO] nebula-metad: Running as 25600, Listening on 9559
[INFO] nebula-graphd: Exited
[INFO] nebula-storaged: Running as 25646, Listening on 9779
```

The Nebula Graph services consist of the Meta Service, Graph Service, and Storage Service. The configuration files for all three services are stored in the `/usr/local/nebula/etc/` directory by default. You can check the configuration files according to the returned information to troubleshoot problems.

In docker container (deployed with docker-compose)

Run the following command in the `nebula-docker-compose/` directory to check the service status of Nebula Graph.

```
nebula-docker-compose$ docker-compose ps
          Name           Command           State           Ports
-----+-----+-----+-----+-----+-----+-----+-----+
nebula-docker-compose_graphd1_1   /usr/local/nebula/bin/nebu ...   Up (healthy)   0.0.0.0:49223->19669/tcp, 0.0.0.0:49228->19670/tcp, 0.0.0.0:49224->9669/tcp
tcp
nebula-docker-compose_graphd2_1   /usr/local/nebula/bin/nebu ...   Up (healthy)   0.0.0.0:49229->19669/tcp, 0.0.0.0:49228->19670/tcp, 0.0.0.0:49230->9669/tcp
tcp
nebula-docker-compose_graphd_1    /usr/local/nebula/bin/nebu ...   Up (healthy)   0.0.0.0:49221->19669/tcp, 0.0.0.0:49220->19670/tcp, 0.0.0.0:9669->9669/tcp
tcp
nebula-docker-compose_metad0_1    ./bin/nebula-metad --flagf ...   Up (healthy)   0.0.0.0:49212->19559/tcp, 0.0.0.0:49211->19560/tcp, 0.0.0.0:49213->9559/tcp
tcp,
nebula-docker-compose_metad1_1    ./bin/nebula-metad --flagf ...   Up (healthy)   0.0.0.0:49209->19559/tcp, 0.0.0.0:49208->19560/tcp, 0.0.0.0:49210->9559/tcp
tcp,
nebula-docker-compose_metad2_1    ./bin/nebula-metad --flagf ...   Up (healthy)   0.0.0.0:49206->19559/tcp, 0.0.0.0:49205->19560/tcp, 0.0.0.0:49207->9559/tcp
tcp,
nebula-docker-compose_storaged0_1  ./bin/nebula-storaged --fl ...   Up (healthy)   0.0.0.0:49218->19779/tcp, 0.0.0.0:49217->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:49219->9779/tcp, 9780/tcp
nebula-docker-compose_storaged1_1  ./bin/nebula-storaged --fl ...   Up (healthy)   0.0.0.0:49215->19779/tcp, 0.0.0.0:49214->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:49216->9779/tcp, 9780/tcp
nebula-docker-compose_storaged2_1  ./bin/nebula-storaged --fl ...   Up (healthy)   0.0.0.0:49226->19779/tcp, 0.0.0.0:49225->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:49227->9779/tcp, 9780/tcp
```

To troubleshoot for a specific service, users can first confirm the container name (such as `nebula-docker-compose_graphd2_1`), then run `docker ps` to find the corresponding `CONTAINER ID` (The ID is `2a6c56c405f5` in the example), and use the `CONTAINER ID` to log in the container to troubleshoot problems.

```
nebula-docker-compose$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
PORTS
2a6c56c405f5        vesoft/nebula-graphd:v2-nightly   "/usr/local/nebula/b..."   36 minutes ago   Up 36 minutes (healthy)   0.0.0.0:49230->9669/tcp, 0.0.0.0:49229->19669/tcp, 0.0.0.0:49228->19670/tcp
7042e0a8e83d        vesoft/nebula-storaged:v2-nightly   "./bin/nebula-storag..."   36 minutes ago   Up 36 minutes (healthy)   9777-9778/tcp, 9780/tcp, 0.0.0.0:49227->9779/tcp, 0.0.0.0:49226->19779/tcp, 0.0.0.0:49225->19780/tcp
18e3ea63ad65        vesoft/nebula-storaged:v2-nightly   "./bin/nebula-storag..."   36 minutes ago   Up 36 minutes (healthy)   9777-9778/tcp, 9780/tcp, 0.0.0.0:49219->9779/tcp, 9780/tcp
4dcabfe8677a        vesoft/nebula-graphd:v2-nightly   "/usr/local/nebula/b..."   36 minutes ago   Up 36 minutes (healthy)   0.0.0.0:49224->9669/tcp, 0.0.0.0:49223->19669/tcp, 0.0.0.0:49222->19670/tcp
a74054c6ae25        vesoft/nebula-graphd:v2-nightly   "/usr/local/nebula/b..."   36 minutes ago   Up 36 minutes (healthy)   0.0.0.0:9669->9669/tcp, 0.0.0.0:49221->19669/tcp, 0.0.0.0:49220->19670/tcp
880025a3858c        vesoft/nebula-storaged:v2-nightly   "./bin/nebula-storag..."   36 minutes ago   Up 36 minutes (healthy)   9777-9778/tcp, 9780/tcp, 0.0.0.0:49216->9779/tcp, 0.0.0.0:49215->19779/tcp, 0.0.0.0:49214->19780/tcp
45736a32a23a        vesoft/nebula-metad:v2-nightly    "./bin/nebula-metad ..."  36 minutes ago   Up 36 minutes (healthy)   9560/tcp, 0.0.0.0:49213->9559/tcp, 0.0.0.0:49212->19559/tcp, 0.0.0.0:49211->19560/tcp
3b2c90eb073e        vesoft/nebula-metad:v2-nightly    "./bin/nebula-metad ..."  36 minutes ago   Up 36 minutes (healthy)   9560/tcp, 0.0.0.0:49207->9559/tcp, 0.0.0.0:49206->19559/tcp, 0.0.0.0:49205->19560/tcp
7bb31b7a5b3f        vesoft/nebula-metad:v2-nightly    "./bin/nebula-metad ..."  36 minutes ago   Up 36 minutes (healthy)   9560/tcp, 0.0.0.0:49210->9559/tcp, 0.0.0.0:49209->19559/tcp, 0.0.0.0:49208->19560/tcp

nebula-docker-compose$ docker exec -it 2a6c56c405f5 bash
[root@2a6c56c405f5 nebula]#
```

3.2.5 Next

[Connect to Nebula Graph](#)

Last update: August 27, 2021

3.3 Connect to Nebula Graph

Nebula Graph supports multiple types of clients, including a CLI client, a GUI client, and clients developed in popular programming languages. This topic provides an overview of Nebula Graph clients and basic instructions on how to use the native CLI client, Nebula Console.

3.3.1 Nebula Graph clients

You can use supported [clients](#) or [console](#) to connect to Nebula Graph database.

3.3.2 Use Nebula Console to connect to Nebula Graph

Prerequisites

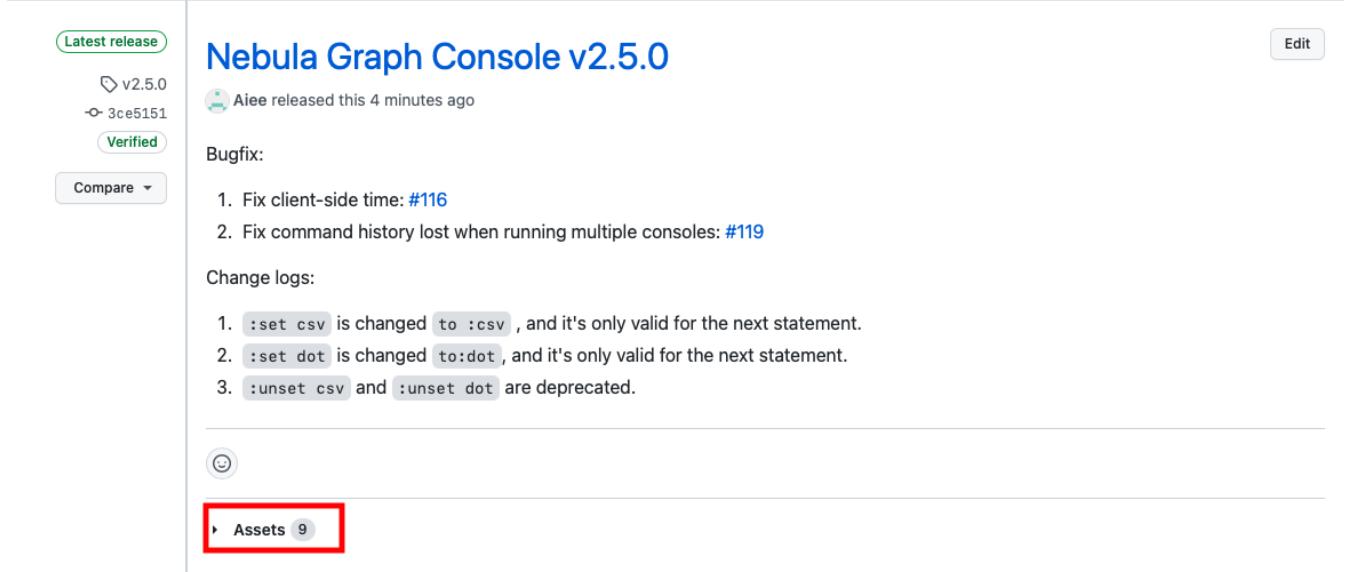
- Nebula Graph services are started. For how to start services, see [Start and Stop Nebula Graph](#).
- The machines that run Nebula Console and Nebula Graph services have network access to each other.

Steps

1. On the [nebula-console](#) page, select a Nebula Console version and click **Assets**.

Note

We recommend that you select the **latest** release.



The screenshot shows a software release page for 'Nebula Graph Console v2.5.0'. The page includes the following details:

- Latest release:** v2.5.0 (3ce5151)
- Verifier:** Verified
- Compare:** Compare dropdown
- Release Note:** Aiee released this 4 minutes ago
- Edit:** Edit button
- Bugfix:**
 - 1. Fix client-side time: #116
 - 2. Fix command history lost when running multiple consoles: #119
- Change logs:**
 - 1. `:set csv` is changed to `:csv`, and it's only valid for the next statement.
 - 2. `:set dot` is changed to `:dot`, and it's only valid for the next statement.
 - 3. `:unset csv` and `:unset dot` are deprecated.
- Assets:** A button with a red box around it, indicating the target for the user's next step.

2. In the **Assets** area, find the correct binary file for the machine where you want to run Nebula Console and download the file to the machine.

Assets 9

 nebula-console-darwin-amd64-v2.5.0	4.04 MB
 nebula-console-darwin-arm64-v2.5.0	3.96 MB
 nebula-console-linux-amd64-v2.5.0	4.04 MB
 nebula-console-linux-arm-v2.5.0	3.61 MB
 nebula-console-linux-arm64-v2.5.0	3.83 MB
 nebula-console-windows-amd64-v2.5.0.exe	3.99 MB
 nebula-console-windows-arm-v2.5.0.exe	3.49 MB
 Source code (zip)	
 Source code (tar.gz)	

3. (Optional) Rename the binary file to `nebula-console` for convenience.

 Note

For Windows, rename the file to `nebula-console.exe`.

4. On the machine that runs Nebula Console, run the following command to grant the execution privileges of the `nebula-console` binary file to the user.

 Note

For Windows, skip this step.

```
$ chmod 111 nebula-console
```

5. In the command line interface, change the working directory to the one where the `nebula-console` binary file is stored.

6. Run the following command to connect to Nebula Graph.

- For Linux or macOS:

```
$ ./nebula-console -addr <ip> -port <port> -u <username> -p <password>
[-t 120] [-e "nGQL_statement" | -f filename.nGQL]
```

- For Windows:

```
> nebula-console.exe -addr <ip> -port <port> -u <username> -p <password>
[-t 120] [-e "nGQL_statement" | -f filename.nGQL]
```

The descriptions are as follows.

Parameter	Description
-h	Shows the help menu.
-addr	Sets the IP address of the graphd service. The default address is 127.0.0.1.
-port	Sets the port number of the graphd service. The default port number is 9669.
-u/-user	Sets the username of your Nebula Graph account. Before enabling authentication, you can use any existing username (The default username is <code>root</code>).
-p/-password	Sets the password of your Nebula Graph account. Before enabling authentication, you can use any characters as the password.
-t/-timeout	Sets an integer-type timeout threshold of the connection. The unit is second. The default value is 120.
-e/-eval	Sets a string-type nGQL statement. The nGQL statement will be executed once the connection succeeds. The connection stops after the result is returned.
-f/-file	Sets the path of an nGQL file. The nGQL statements in the file are executed once the connection succeeds. The connection stops after the result is returned.

Users can use the command of `./nebula-console --help` to get the details of all the parameters and can also find more details in the [Nebula Console Repository](#).

3.3.3 Nebula Console commands

Nebula Console supports some commands. Users can export all the query results into a CSV file or a DOT file, and import test datasets.

Note

- The commands are case-insensitive.

Export CSV files

Note

- The CSV file is stored in the working directory. Run the Linux command `pwd` to show the working directory.
- This command only takes effect on the next query statement.

The command is as follows.

```
nebula> :CSV <file_name.csv>
```

Export DOT files

Note

- The DOT file is stored in the working directory. Run the Linux command `pwd` to show the working directory.
- The content of the DOT file can be copied and pasted in the [GraphvizOnline](#) to generate a visual execution plan diagram.
- This command only takes effect on the next query statement.

The command is as follows.

```
nebula> :dot <file_name.dot>
```

The example is as follows.

```
nebula> :dot a.dot
nebula> PROFILE FORMAT="dot" GO FROM "player100" OVER follow;
```

Load test datasets

The name of the test dataset is nba. To check the detailed information on the Schema and data, run the relevant `SHOW` command.

The command is as follows.

```
nebula> :play nba
```

Repeat the execution

This command repeats the next command N times, and then prints the average execution time. The command is as follows.

```
nebula> :repeat N
```

Examples are as follows.

```
nebula> :repeat 3
nebula> GO FROM "player100" OVER follow;
+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
| "player125" |
+-----+
Got 2 rows (time spent 2602/3214 us)

Fri, 20 Aug 2021 06:36:05 UTC

+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
| "player125" |
+-----+
Got 2 rows (time spent 583/849 us)

Fri, 20 Aug 2021 06:36:05 UTC

+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
| "player125" |
+-----+
Got 2 rows (time spent 496/671 us)

Fri, 20 Aug 2021 06:36:05 UTC

Executed 3 times, (total time spent 3681/4734 us), (average time spent 1227/1578 us)
```

Sleep

The command sleeps for N seconds. It is often used to modify the Schema because the modification of the Schema is implemented asynchronously, and Nebula Graph synchronizes the data in the next heartbeat cycle. The command is as follows:

```
nebula> :sleep N
```

3.3.4 Disconnect Nebula Console from Nebula Graph

Users can use `:EXIT` or `:QUIT` to disconnect from Nebula Graph. For convenience, Nebula Console supports using these commands in lower case without the colon, such as `quit`.

The example is as follows.

```
nebula> :QUIT
Bye root!
```

3.3.5 FAQ

How can I install Nebula Console from the source code?

To download and compile the latest source code of Nebula Console, follow the instructions on [GitHub Nebula Console](#).

Last update: August 27, 2021

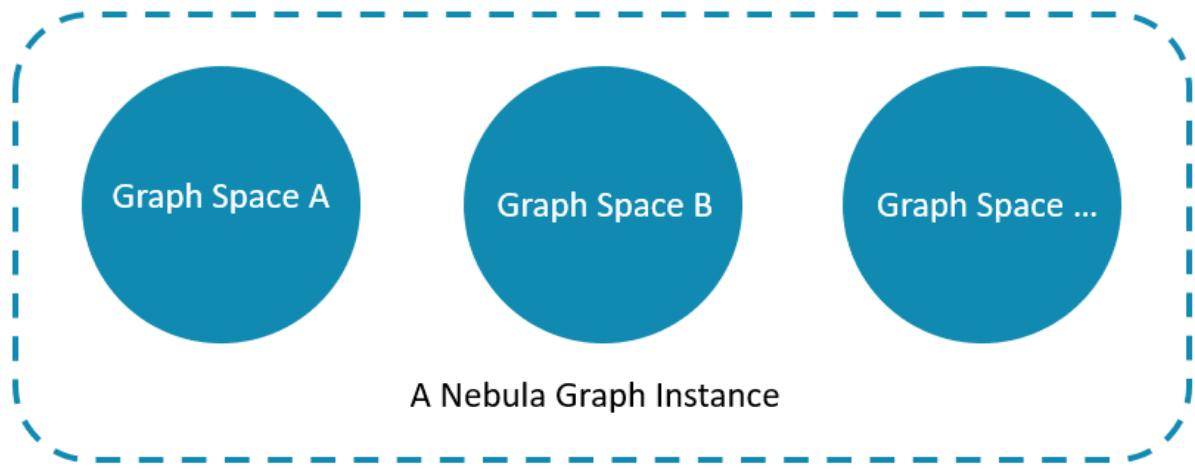
3.4 Nebula Graph CRUD

This topic will describe the basic CRUD operations in Nebula Graph.

For more information, see [nGQL guide](#).

3.4.1 Graph space and Nebula Graph schema

A Nebula Graph instance consists of one or more graph spaces. Graph spaces are physically isolated from each other. You can use different graph spaces in the same instance to store different datasets.

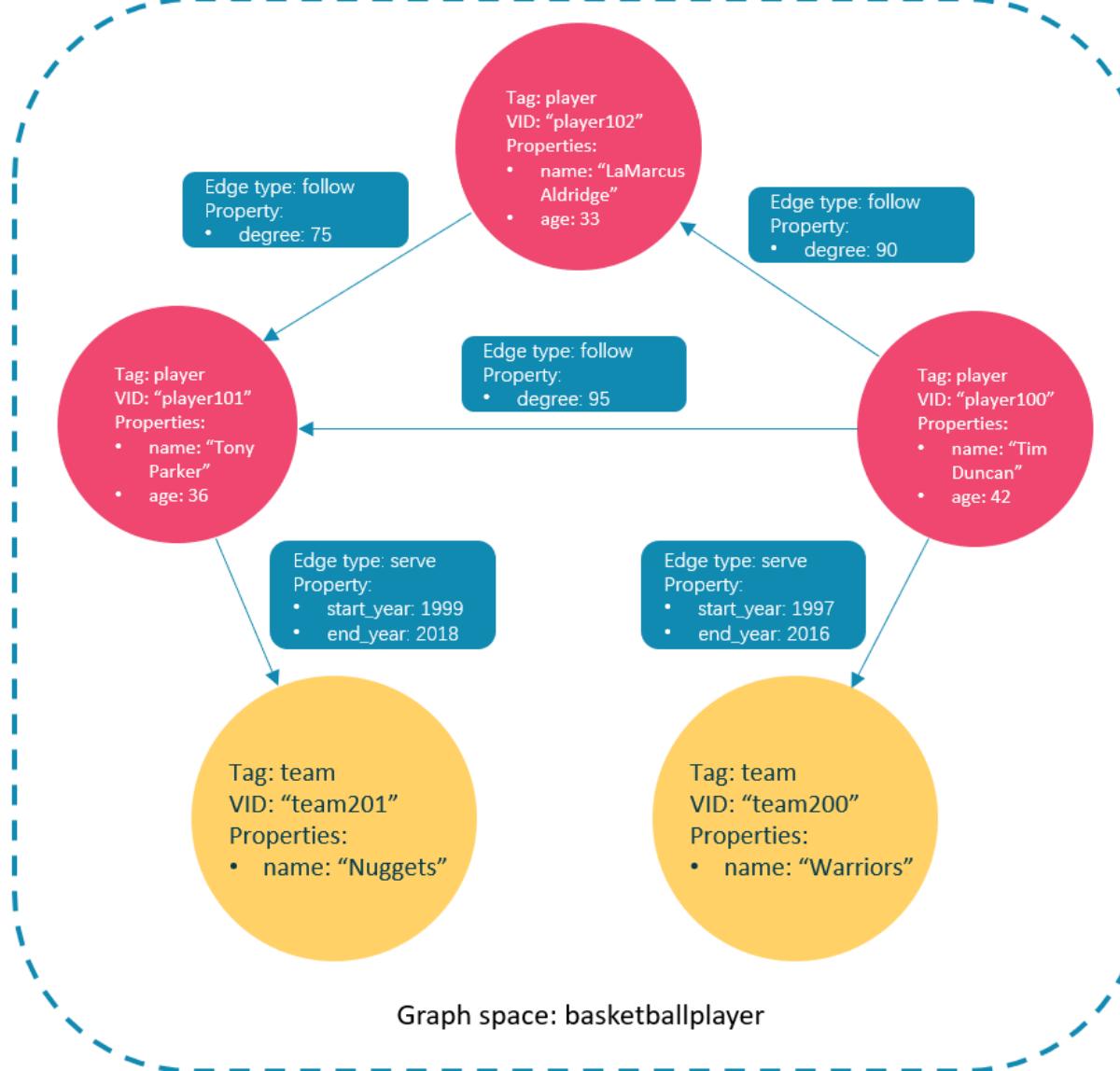


To insert data into a graph space, define a schema for the graph database. Nebula Graph schema is based on the following components.

Schema component	Description
Vertex	Represents an entity in the real world. A vertex can have one or more tags.
Tag	The type of the same group of vertices. It defines a set of properties that describes the types of vertices.
Edge	Represents a directed relationship between two vertices.
Edge type	The type of an edge. It defines a group of properties that describes the types of edges.

For more information, see [Data modeling](#).

In this topic, we will use the following dataset to demonstrate basic CRUD operations.



3.4.2 Check the machine status in the Nebula Graph cluster

Note

First, we recommend that you check the machine status to make sure that all the Storage services are connected to the Meta services. Run `SHOW HOSTS` as follows.

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" |
+-----+-----+-----+-----+-----+
| "storaged1" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" |
+-----+-----+-----+-----+-----+
| "storaged2" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" |
+-----+-----+-----+-----+-----+
| "Total" | __EMPTY__ | __EMPTY__ | 0 | __EMPTY__ | __EMPTY__ |
+-----+-----+-----+-----+-----+
Got 4 rows (time spent 1061/2251 us)
```

From the **Status** column of the table in the return results, you can see that all the Storage services are online.

Asynchronous implementation of creation and alteration

⚠ Caution

Nebula Graph implements the following creation or alteration operations asynchronously in the **next** heartbeat cycle. The operations will not take effect until they finish.

- CREATE SPACE
- CREATE TAG
- CREATE EDGE
- ALTER TAG
- ALTER EDGE
- CREATE TAG INDEX
- CREATE EDGE INDEX

>Note

The default heartbeat interval is 10 seconds. To change the heartbeat interval, modify the `heartbeat_interval_secs` parameter in the [configuration files](#) for all services.

To make sure the follow-up operations work as expected, take one of the following approaches:

- Run `SHOW` or `DESCRIBE` statements accordingly to check the status of the objects, and make sure the creation or alteration is complete. If it is not, wait a few seconds and try again.
- Wait for two heartbeat cycles, i.e., 20 seconds.

3.4.3 Create and use a graph space

nGQL syntax

- Create a graph space:

```
CREATE SPACE [IF NOT EXISTS] <graph_space_name> (
  [partition_num = <partition_number>],
  [replica_factor = <replica_number>],
  vid_type = {FIXED_STRING(<N>) | INT64}
)
[COMMENT = '<comment>'];
```

For more information on parameters, see [CREATE SPACE](#).

- List graph spaces and check if the creation is successful:

```
nebula> SHOW SPACES;
```

- Use a graph space:

```
USE <graph_space_name>;
```

Examples

1. Use the following statement to create a graph space named `basketballplayer`.

```
nebula> CREATE SPACE basketballplayer(partition_num=15, replica_factor=1, vid_type=fixed_string(30));
Execution succeeded (time spent 2817/3280 us)
```

2. Check the partition distribution with `SHOW HOSTS` to make sure that the partitions are distributed in a balanced way.

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 5 | "basketballplayer:5" | "basketballplayer:5" |
+-----+-----+-----+-----+-----+
| "storaged1" | 9779 | "ONLINE" | 5 | "basketballplayer:5" | "basketballplayer:5" |
+-----+-----+-----+-----+-----+
| "storaged2" | 9779 | "ONLINE" | 5 | "basketballplayer:5" | "basketballplayer:5" |
+-----+-----+-----+-----+-----+
| "Total" | | | 15 | "basketballplayer:15" | "basketballplayer:15" |
+-----+-----+-----+-----+-----+
Got 4 rows (time spent 1633/2867 us)
```

If the **Leader distribution** is uneven, use `BALANCE LEADER` to redistribute the partitions. For more information, see [BALANCE](#).

3. Use the `basketballplayer` graph space.

```
nebula[(none)]> USE basketballplayer;
Execution succeeded (time spent 1229/2318 us)
```

You can use `SHOW SPACES` to check the graph space you created.

```
nebula> SHOW SPACES;
+-----+
| Name |
+-----+
| "basketballplayer" |
+-----+
Got 1 rows (time spent 977/2000 us)
```

3.4.4 Create tags and edge types

nGQL syntax

```
CREATE {TAG | EDGE} {<tag_name> | <edge_type>}(<property_name> <data_type>
[, <property_name> <data_type> ...])
[COMMENT = '<comment>'];
```

For more information on parameters, see [CREATE TAG](#) and [CREATE EDGE](#).

Examples

Create tags `player` and `team`, edge types `follow` and `serve`. Descriptions are as follows.

Component name	Type	Property
player	Tag	name (string), age (int)
team	Tag	name (string)
follow	Edge type	degree (int)
serve	Edge type	start_year (int), end_year (int)

```
nebula> CREATE TAG player(name string, age int);
Execution succeeded (time spent 20708/22071 us)

nebula> CREATE TAG team(name string);
Execution succeeded (time spent 5643/6810 us)

nebula> CREATE EDGE follow(degree int);
Execution succeeded (time spent 12665/13934 us)

nebula> CREATE EDGE serve(start_year int, end_year int);
Execution succeeded (time spent 5858/6870 us)
```

3.4.5 Insert vertices and edges

Users can use the `INSERT` statement to insert vertices or edges based on existing tags or edge types.

nGQL syntax

- Insert vertices:

```
INSERT VERTEX [IF NOT EXISTS] <tag_name> (<property_name>[, <property_name>...])
[, <tag_name> <property_name>[, <property_name>...]], ...
{VALUES | VALUE} <vid>: (<property_value>[, <property_value>...])
[, <vid>: (<property_value>[, <property_value>...]);
```

`VID` is short for Vertex ID. A `VID` must be a unique string value in a graph space. For details, see [INSERT VERTEX](#).

- Insert edges:

```
INSERT EDGE [IF NOT EXISTS] <edge_type> (<property_name>[, <property_name>...])
{VALUES | VALUE} <src_vid> -> <dst_vid>[@<rank>] : (<property_value>[, <property_value>...])
[, <src_vid> -> <dst_vid>[@<rank>] : (<property_name>[, <property_name>...]), ...];
```

For more information on parameters, see [INSERT EDGE](#).

Examples

- Insert vertices representing basketball players and teams:

```
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42);
Execution succeeded (time spent 28196/30896 us)

nebula> INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36);
Execution succeeded (time spent 2708/3834 us)

nebula> INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
Execution succeeded (time spent 1945/3294 us)
```

```
nebula> INSERT VERTEX team(name) VALUES "team200":("Warriors"), "team201":("Nuggets");
Execution succeeded (time spent 2269/3310 us)
```

- Insert edges representing the relations between basketball players and teams:

```
nebula> INSERT EDGE follow(degree) VALUES "player100" -> "player101":(95);
Execution succeeded (time spent 3362/4542 us)

nebula> INSERT EDGE follow(degree) VALUES "player100" -> "player102":(90);
Execution succeeded (time spent 2974/4274 us)

nebula> INSERT EDGE follow(degree) VALUES "player102" -> "player101":(75);
Execution succeeded (time spent 1891/3096 us)

nebula> INSERT EDGE serve(start_year, end_year) VALUES "player100" -> "team200":(1997, 2016), "player101" -> "team201":(1999, 2018);
Execution succeeded (time spent 6064/7104 us)
```

3.4.6 Read data

- The `GO` statement can traverse the database based on specific conditions. A `GO` traversal starts from one or more vertices, along one or more edges, and returns information in a form specified in the `YIELD` clause.
- The `FETCH` statement is used to get properties from vertices or edges.
- The `LOOKUP` statement is based on `indexes`. It is used together with the `WHERE` clause to search for the data that meet the specific conditions.
- The `MATCH` statement is the most commonly used statement for graph data querying. It can describe all kinds of graph patterns, but it relies on `indexes` to match data patterns in Nebula Graph. Therefore, its performance still needs optimization.

nGQL syntax

- GO

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

- FETCH

- Fetch properties on tags:

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

- Fetch properties on edges:

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

- LOOKUP

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

- MATCH

```
MATCH <pattern> [<WHERE clause>] RETURN <output>;
```

Examples of `GO` statement

- Search for the players that the player with VID `player100` follows.

```
nebula> GO FROM "player100" OVER follow;
+-----+
| follow._dst |
+-----+
```

```
| "player101" |
+-----+
| "player102" |
+-----+
Got 2 rows (time spent 12097/14220 us)
```

- Filter the players that the player with VID `player100` follows whose age is equal to or greater than 35. Rename the corresponding columns in the results with `Teammate` and `Age`.

```
nebula> GO FROM "player100" OVER follow WHERE $$ .player.age >= 35 \
    YIELD $$ .player.name AS Teammate, $$ .player.age AS Age;
+-----+-----+
| Teammate | Age |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
Got 1 rows (time spent 8206/9335 us)
```

Clause/Sign	Description
<code>YIELD</code>	Specifies what values or results you want to return from the query.
<code>\$\$</code>	Represents the target vertices.
<code>\</code>	A line-breaker.

- Search for the players that the player with VID `player100` follows. Then Retrieve the teams of the players that the player with VID `player100` follows. To combine the two queries, use a pipe or a temporary variable.

- With a pipe:

```
nebula> GO FROM "player100" OVER follow YIELD follow._dst AS id | \
    GO FROM $^.id OVER serve YIELD $$ .team.name AS Team, \
    $$ .player.name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| "Nuggets" | "Tony Parker" |
+-----+-----+
Got 1 rows (time spent 5055/8203 us)
```

Clause/Sign	Description
<code>\$^</code>	Represents the source vertex of the edge.
<code> </code>	A pipe symbol can combine multiple queries.
<code>\$-</code>	Represents the outputs of the query before the pipe symbol.

- With a temporary variable:

Note

Once a composite statement is submitted to the server as a whole, the life cycle of the temporary variables in the statement ends.

```
nebula> $var = GO FROM "player100" OVER follow YIELD follow._dst AS id; \
    GO FROM $var.id OVER serve YIELD $$ .team.name AS Team, \
    $$ .player.name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| Nuggets | Tony Parker |
+-----+-----+
Got 1 rows (time spent 3103/3711 us)
```

Example of `FETCH` statement

Use `FETCH`: Fetch the properties of the player with VID `player100`.

```
nebula> FETCH PROP ON player "player100";
+-----+
| vertices_ |
+-----+
```

```
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+
Got 1 rows (time spent 2006/2406 us)
```

Note

The examples of `LOOKUP` and `MATCH` statements are in [indexes](#).

3.4.7 Update vertices and edges

Users can use the `UPDATE` or the `UPSERT` statements to update existing data.

`UPSERT` is the combination of `UPDATE` and `INSERT`. If you update a vertex or an edge with `UPSERT`, the database will insert a new vertex or edge if it does not exist.

Note

`UPSERT` operates serially in a partition-based order. Therefore, it is slower than `INSERT OR UPDATE`. And `UPSERT` has concurrency only between multiple partitions.

nGQL syntax

- `UPDATE` vertices:

```
UPDATE VERTEX <vid> SET <properties to be updated>
[WHEN <condition>] [YIELD <columns>];
```

- `UPDATE` edges:

```
UPDATE EDGE <source vid> -> <destination vid> [@rank] OF <edge_type>
SET <properties to be updated> [WHEN <condition>] [YIELD <columns to be output>];
```

- `UPSERT` vertices or edges:

```
UPSERT {VERTEX <vid> | EDGE <edge_type>} SET <update_columns>
[WHEN <condition>] [YIELD <columns>];
```

Examples

- `UPDATE` the `name` property of the vertex with VID `player100` and check the result with the `FETCH` statement.

```
nebula> UPDATE VERTEX "player100" SET player.name = "Tim";
Execution succeeded (time spent 3483/3914 us)

nebula> FETCH PROP ON player "player100";
+-----+
| vertices_          |
+-----+
| ("player100" :player{age: 42, name: "Tim"}) |
```

```
+-----+
Got 1 rows (time spent 2463/3042 us)
```

- `UPDATE` the `degree` property of an edge and check the result with the `FETCH` statement.

```
nebula> UPDATE EDGE "player100" -> "player101" OF follow SET degree = 96;
Execution succeeded (time spent 3932/4432 us)

nebula> FETCH PROP ON follow "player100" -> "player101";
+-----+
| edges_
| +-----+
| | [:follow "player100" ->"player101" @0 {degree: 96}] |
| +-----+
Got 1 rows (time spent 2205/2800 us)
```

- Insert a vertex with VID `player111` and `UPSERT` it.

```
nebula> INSERT VERTEX player(name, age) VALUES "player111":("Ben Simmons", 22);
Execution succeeded (time spent 2115/2900 us)

Wed, 21 Oct 2020 11:11:50 UTC

nebula> UPSERT VERTEX "player111" SET player.name = "Dwight Howard", player.age = $^.player.age + 11 \
  WHEN $^.player.name == "Ben Simmons" AND $^.player.age > 20 \
  YIELD $^.player.name AS Name, $^.player.age AS Age;
+-----+
| Name      | Age |
+-----+-----+
| Dwight Howard | 33 |
+-----+
Got 1 rows (time spent 1815/2329 us)
```

3.4.8 Delete vertices and edges

nGQL syntax

- Delete vertices:

```
DELETE VERTEX <vid1>[, <vid2>...]
```

- Delete edges:

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

Examples

- Delete vertices:

```
nebula> DELETE VERTEX "team1", "team2";
Execution succeeded (time spent 4337/4782 us)
```

- Delete edges:

```
nebula> DELETE EDGE follow "team1" -> "team2";
Execution succeeded (time spent 3700/4101 us)
```

3.4.9 About indexes

Users can add indexes to tags and edge types with the [CREATE INDEX](#) statement.

⚠ Must-read for using indexes

Both `MATCH` and `LOOKUP` statements depend on the indexes. But indexes can dramatically reduce the write performance (as much as 90% or even more). **DO NOT** use indexes in production environments unless you are fully aware of their influences on your service.

Users **MUST** rebuild indexes for pre-existing data. Otherwise, the pre-existing data cannot be indexed and therefore cannot be returned in `MATCH` or `LOOKUP` statements. For more information, see [REBUILD INDEX](#).

nGQL syntax

- Create an index:

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

- Rebuild an index:

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

Examples

Create and rebuild indexes for the `name` property on all vertices with the tag `player`.

```
nebula> CREATE TAG INDEX player_index_0 on player(name(20));
nebula> REBUILD TAG INDEX player_index_0;
```

Note

Define the index length when creating an index for a variable-length property. In UTF-8 encoding, a non-ascii character occupies 3 bytes. You should set an appropriate index length according to the variable-length property. For example, the index should be 30 bytes for 10 non-ascii characters. For more information, see [CREATE INDEX](#)

Examples of LOOKUP and MATCH (index-based)

Make sure there is an `index` for `LOOKUP` or `MATCH` to use. If there is not, create an index first.

Find the information of the vertex with the tag `player` and its value of the `name` property is `Tony Parker`.

This example creates the index `player_name_0` on the player name property.

```
nebula> CREATE TAG INDEX player_name_0 on player(name(10));
Execution succeeded (time spent 3465/4150 us)
```

This example rebuilds the index to make sure it takes effect on pre-existing data.

```
nebula> REBUILD TAG INDEX player_name_0
+-----+
| New Job Id |
+-----+
| 31          |
+-----+
Got 1 rows (time spent 2379/3033 us)
```

This example uses the `LOOKUP` statement to retrieve the vertex property.

```
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
    YIELD player.name, player.age;
+-----+-----+-----+
| VertexID | player.name | player.age |
+-----+-----+-----+
| "player101" | "Tony Parker" | 36          |
+-----+-----+-----+
```

This example uses the `MATCH` statement to retrieve the vertex property.

```
nebula> MATCH (v:player{name:"Tony Parker"}) RETURN v;
+-----+
| v          |
+-----+
| {"player101":player{age: 36, name: "Tony Parker"} } |
+-----+
Got 1 rows (time spent 5132/6246 us)
```

Last update: August 27, 2021

4. nGQL guide

4.1 nGQL overview

4.1.1 Nebula Graph Query Language (nGQL)

This topic gives an introduction to the query language of Nebula Graph, nGQL.

What is nGQL

nGQL is a declarative graph query language for Nebula Graph. It allows expressive and efficient [graph patterns](#). nGQL is designed for both developers and operations professionals. nGQL is an SQL-like query language, so it's easy to learn.

nGQL is a project in progress. New features and optimizations are done steadily. There can be differences between syntax and implementation. Submit an [issue](#) to inform the Nebula Graph team if you find a new issue of this type. Nebula Graph 2.0 or later versions will support [openCypher 9](#).

What can nGQL do

- Supports graph traversals
- Supports pattern match
- Supports aggregation
- Supports graph mutation
- Supports access control
- Supports composite queries
- Supports index
- Supports most openCypher 9 graph query syntax (but mutations and controls syntax are not supported)

Example data Basketballplayer

You can download the example data [Basketballplayer](#) in Nebula Graph. After downloading the example data, you can import it to Nebula Graph by using the `-f` option in [Nebula Graph Console](#).

Placeholder identifiers and values

Refer to the following standards in nGQL:

- (Draft) ISO/IEC JTC1 N14279 SC 32 - Database_Languages - GQL
- (Draft) ISO/IEC JTC1 SC32 N3228 - SQL_Property_Graph_Queries - SQLPGQ
- OpenCypher 9

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 syntax, see 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

For example, create vertices or edges in nGQL syntax:

```
CREATE {TAG | EDGE} {<tag_name> | <edge_type>}(<property_name> <data_type>
[, <property_name> <data_type> ...]);
```

Example statement:

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

About openCypher compatibility

NATIVE NGQL AND OPENCYCIPHER

Native nGQL is the part of a graph query language designed and implemented by Nebula Graph. OpenCypher is a graph query language maintained by openCypher Implementers Group.

The latest version is openCypher 9. The compatible parts of openCypher in nGQL are called openCypher compatible sentences (short as openCypher).

Note

```
nGQL = native nGQL + openCypher compatible sentences
```

IS NGQL COMPATIBLE WITH OPENCYCIPHER 9?

nGQL is partially compatible with openCypher 9

37 known incompatible items are listed in [Nebula Graph Issues](#). Submit an issue with the `incompatible` tag if you find a new issue of this type. You can search in this manual with the keyword `compatibility` to find major compatibility issues.

WHAT ARE THE MAJOR DIFFERENCES BETWEEN NGQL AND OPENCYpher 9?

The following are some major differences (by design incompatible) between nGQL and openCypher.

Category	openCypher 9	nGQL
Schema	Optional Schema	Strong Schema
Equality operator	=	==
Math exponentiation	^	^ not supported. Use pow(x, y) instead.
Edge rank	no such concept	edge rank (reference by @)
Statement	-	All DMLs (CREATE, MERGE, etc) of openCypher 9, and OPTIONAL MATCH are not supported.
Label and tag	A label is used for searching a vertex, namely an index of vertex.	A tag defines the type of a vertex and its corresponding properties. It cannot be used as an index.
Pre-compiling and parameterized query	support	not supported

Compatibility

OpenCypher 9 and Cypher have some differences in grammar and licence. For example,

1. Cypher requires that **All Cypher statements are explicitly run within a transaction**. While openCypher has no such requirement. And nGQL does not support transactions.
2. Cypher has a variety of constraints, including Unique node property constraints, Node property existence constraints, Relationship property existence constraints, and Node key constraints. While OpenCypher has no such constraints. As a strong schema system, most of the constraints mentioned above can be solved through schema definitions (including NOT NULL) in nGQL. The only function that cannot be supported is the UNIQUE constraint.
3. Cypher has APoC, while openCypher 9 does not have APoC. Cypher has Blot protocol support requirements, while openCypher 9 does not.

WHERE CAN I FIND MORE NGQL EXAMPLES?

Find more than 2500 nGQL examples in the [features](#) directory on the Nebula Graph GitHub page.

The `features` directory consists of `.feature` files. Each file records scenarios that you can use as nGQL examples. Here is an example:

```
Feature: Basic match

Background:
  Given a graph with space named "basketballplayer"

Scenario: Single node
  When executing query:
  """
    MATCH (v:player {name: "Yao Ming"}) RETURN v;
  """
  Then the result should be, in any order, with relax comparison:
  | v
  | ("player133" :player{age: 38, name: "Yao Ming"}) |

Scenario: One step
  When executing query:
  """
    MATCH (v1:player{name: "LeBron James"}) -[r]-> (v2)
    RETURN type(r) AS Type, v2.name AS Name
  """
  Then the result should be, in any order:
  | Type    | Name      |
  | "follow" | "Ray Allen" |
  | "serve"  | "Lakers"   |
  | "serve"  | "Heat"     |
  | "serve"  | "Cavaliers" |
```

```

Feature: Comparison of where clause

Background:
  Given a graph with space named "basketballplayer"

Scenario: push edge props filter down
  When profiling query:
  """
  GO FROM "player100" OVER follow
  WHERE follow.degree IN [v IN [95,99] WHERE v > 0]
  YIELD follow._dst, follow.degree
  """
  Then the result should be, in any order:
  | follow._dst | follow.degree |
  | "player101" | 95 |
  | "player125" | 95 |
  And the execution plan should be:
  | id | name | dependencies | operator info |
  | 0 | Project | 1 | |
  | 1 | GetNeighbors | 2 | {"filter": "(follow.degree IN [v IN [95,99] WHERE (v>0)])"} |
  | 2 | Start | | |

```

The keywords in the preceding example are described as follows.

Keyword	Description
Feature	Describes the topic of the current <code>.feature</code> file.
Background	Describes the background information of the current <code>.feature</code> file.
Given	Describes the prerequisites of running the test statements in the current <code>.feature</code> file.
Scenario	Describes the scenarios. If there is the <code>@skip</code> before one <code>Scenario</code> , this scenario may not work and do not use it as a working example in a production environment.
When	Describes the nGQL statement to be executed. It can be a <code>executing query</code> or <code>profiling query</code> .
Then	Describes the expected return results of running the statement in the <code>When</code> clause. If the return results in your environment do not match the results described in the <code>.feature</code> file, submit an issue to inform the Nebula Graph team.
And	Describes the side effects of running the statement in the <code>When</code> clause.
<code>@skip</code>	This test case will be skipped. Commonly, the to-be-tested code is not ready.

Welcome to [add more tck case](#) and return automatically to the using statements in CI/CD.

DOES IT SUPPORT TINKERPOP GREMLIN?

No. And no plan to support that.

DOES NEBULA GRAPH SUPPORT W3C RDF (SPARQL) OR GRAPHQL?

No. And no plan to support that.

The data model of Nebula Graph is the property graph. And as a strong schema system, Nebula Graph does not support RDF.

Nebula Graph Query Language does not support `SPARQL` nor `GraphQL`.

Last update: August 24, 2021

4.1.2 Patterns

Patterns and graph pattern matching are the very heart of a graph query language. This topic will describe the patterns in Nebula Graph.

Patterns for vertices

A vertex is described using a pair of parentheses and is typically given a name. For example:

(a)

This simple pattern describes a single vertex and names that vertex using the variable `a`.

Patterns for related vertices

A more powerful construct is a pattern that describes multiple vertices and edges between them. Patterns describe an edge by employing an arrow between two vertices. For example:

(a)-[]->(b)

This pattern describes a very simple data structure: two vertices and a single edge from one to the other. In this example, the two vertices are named as `a` and `b` respectively and the edge is `directed`: it goes from `a` to `b`.

This manner of describing vertices and edges can be extended to cover an arbitrary number of vertices and the edges between them, for example:

(a)-[]->(b)-[]-(c)

Such a series of connected vertices and edges is called a `path`.

Note that the naming of the vertices in these patterns is only necessary when one needs to refer to the same vertex again, either later in the pattern or elsewhere in the query. If not, the name may be omitted as follows:

(a)-[]->()-[]-(c)

Patterns for tags

OpenCypher compatibility

The concept of `tag` in nGQL has a few differences from that of `label` in openCypher. For example, you must create a `tag` before using it. And a `tag` also defines the type of properties.

In addition to simply describing the vertices in the graphs, patterns can also describe the tags of the vertices. For example:

(a:User)-[]->(b)

Patterns can also describe a vertex that has multiple tags. For example:

(a:User:Admin)-[]->(b)

OpenCypher compatibility

The `MATCH` statement in nGQL does not support matching multiple tags with `(a:User:Admin)`. If you need to match multiple tags, use filtering conditions, such as `WHERE "User" IN tags(n) AND "Admin" IN tags(n)`.

Patterns for properties

Vertices and edges are the fundamental elements in a graph. In nGQL, properties are added to them for richer models.

In the patterns, the properties can be expressed as follows: some key-value pairs are enclosed in curly brackets and separated by commas. For example, a vertex with two properties will be like:

```
(a {name: 'Andres', sport: 'Brazilian Ju-Jitsu'})
```

One of the edges that connect to this vertex can be like:

```
(a)-[{}{blocked: false}]->(b)
```

Patterns for edges

The simplest way to describe an edge is by using the arrow between two vertices, as in the previous examples.

You can describe an edge and its direction using the following statement. If you do not care about its direction, the arrowhead can be omitted. For example:

```
(a)-[]-(b)
```

Like vertices, edges can also be named. A pair of square brackets will be used to separate the arrow and the variable will be placed between them. For example:

```
(a)-[r]->(b)
```

Like the tags on vertices, edges can also have types. To describe an edge with a specific type, use the pattern as follows:

```
(a)-[r:REL_TYPE]->(b)
```

An edge can only have one edge type. But if we'd like to describe some data such that the edge could have a set of types, then they can all be listed in the pattern, separating them with the pipe symbol `|` like this:

```
(a)-[r:TYPE1|TYPE2]->(b)
```

Like vertices, the name of an edge can be omitted. For example:

```
(a)-[:REL_TYPE]->(b)
```

Variable-length pattern

Rather than describing a long path using a sequence of many vertex and edge descriptions in a pattern, many edges (and the intermediate vertices) can be described by specifying a length in the edge description of a pattern. For example:

```
(a)-[*2]->(b)
```

The following pattern describes a graph of three vertices and two edges, all in one path (a path of length 2). It is equivalent to:

```
(a)-[]->()-[]->(b)
```

The range of lengths can also be specified. Such edge patterns are called `variable-length edges`. For example:

```
(a)-[*3..5]->(b)
```

The preceding example defines a path with a minimum length of 3 and a maximum length of 5.

It describes a graph of either 4 vertices and 3 edges, 5 vertices and 4 edges, or 6 vertices and 5 edges, all connected in a single path.

The lower bound can be omitted. For example, to describe paths of length 5 or less, use:

```
(a)-[*..5]->(b)
```

Note

The upper bound must be specified. The following are **NOT** accepted.

```
(a)-[*3..]->(b)  
(a)-[*]->(b)
```

Assigning to path variables

As described above, a series of connected vertices and edges is called a `path`. nGQL allows paths to be named using variables. For example:

```
p = (a)-[*3..5]->(b)
```

You can do this in `MATCH` statement.

Last update: August 24, 2021

4.1.3 Comments

This topic will describe the comments in nGQL.

Legacy version compatibility

- In Nebula Graph 1.0, there are four comment styles: `#`, `--`, `//`, `/* */`.
- In Nebula Graph 2.0, `--` represents an [edge pattern](#) and cannot be used as comments.

Examples

```
nebula> # Do nothing in this line
nebula> RETURN 1+1;      # This comment continues to the end of this line.
nebula> RETURN 1+1;      // This comment continues to the end of this line.
nebula> RETURN 1 /* This is an in-line comment. */ + 1 == 2;
nebula> RETURN 11 +
/* Multi-line comment.      \
Use a backslash as a line break.  \
*/ 12;
```

In nGQL statement, the backslash `\` in a line indicates a line break.

OpenCypher compatibility

- In nGQL, you must add a `\` at the end of every line, even in multi-line comments `/* */`.
- In openCypher, there is no need to use a `\` as a line break.

```
/* openCypher style:
The following comment
spans more than
one line */
MATCH (n:label)
RETURN n;
```

```
/* nGQL style: \
The following comment      \
spans more than      \
one line */      \
MATCH (n:tag) \
RETURN n;
```

Last update: July 13, 2021

4.1.4 Identifier case sensitivity

Identifiers are Case-Sensitive

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

```
nebula> CREATE SPACE my_space (vid_type=FIXED_STRING(30));
nebula> use MY_SPACE;
[ERROR (-8)]: SpaceNotFound:
```

Keywords and Reserved Words are Case-Insensitive

The following statements are equivalent since `show` and `spaces` are keywords.

```
nebula> show spaces;
nebula> SHOW SPACES;
nebula> SHOW spaces;
nebula> show SPACES;
```

Functions are Case-Insensitive

Functions are case-insensitive. For example, `count()`, `COUNT()`, and `couNT()` are equivalent.

```
nebula> WITH [NULL, 1, 1, 2, 2] As a \
    UNWIND a AS b \
    RETURN count(b), COUNT(*), couNT(DISTINCT b);
+-----+-----+-----+
| count(b) | COUNT(*) | couNT(DISTINCT b) |
+-----+-----+-----+
| 4       | 5       | 2       |
+-----+-----+-----+
```

Last update: July 14, 2021

4.1.5 Keywords

Keywords have significance in nGQL. It can be classified into reserved keywords and non-reserved keywords.

Non-reserved keywords are permitted as identifiers without quoting. To use reserved keywords as identifiers, quote them with backticks such as `AND`.

Note

Keywords are case-insensitive.

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

nebula> CREATE TAG `TAG` (name string);
Execution succeeded

nebula> CREATE TAG SPACE(name string);
Execution succeeded
```

- `TAG` is a reserved keyword. To use `TAG` as an identifier, you must quote it with backticks.
- `SPACE` is a non-reserved keyword. You can use it as an identifier without quoting it.

Reserved keywords

```
GO
AS
TO
OR
AND
XOR
USE
SET
FROM
WHERE
MATCH
INSERT
YIELD
RETURN
DESCRIBE
DESC
VERTEX
EDGE
EDGES
UPDATE
UPsert
WHEN
DELETE
FIND
LOOKUP
ALTER
STEPS
STEP
OVER
UPTO
REVERSELY
INDEX
INDEXES
REBUILD
BOOL
INT8
INT16
INT32
INT64
INT
FLOAT
DOUBLE
STRING
FIXED_STRING
TIMESTAMP
DATE
TIME
DATETIME
TAG
TAGS
UNION
INTERSECT
MINUS
```

```
NO
OVERWRITE
SHOW
ADD
CREATE
DROP
REMOVE
IF
NOT
EXISTS
WITH
CHANGE
GRANT
REVOKE
ON
BY
IN
NOT_IN
DOWNLOAD
GET
OF
ORDER
INGEST
COMPACT
FLUSH
SUBMIT
ASC
ASCENDING
DESCENDING
DISTINCT
FETCH
PROP
BALANCE
STOP
LIMIT
OFFSET
IS
NULL
RECOVER
EXPLAIN
PROFILE
FORMAT
CASE
```

Non-reserved keywords

```
HOST
HOSTS
SPACE
SPACES
VALUE
VALUES
USER
USERS
PASSWORD
ROLE
ROLES
GOD
ADMIN
DBA
GUEST
GROUP
PARTITION_NUM
REPLICA_FACTOR
VID_TYPE
CHARSET
COLLATE
COLLATION
ATOMIC_EDGE
ALL
ANY
SINGLE
NONE
REDUCE
LEADER
UUID
DATA
SNAPSHOT
SNAPSHOTS
ACCOUNT
JOBS
JOB
PATH
BIDIRECT
STATS
STATUS
FORCE
PART
PARTS
DEFAULT
HDFS
```

```
CONFIGS
TTL_DURATION
TTL_COL
GRAPH
META
STORAGE
SHORTEST
NOLOOP
OUT
BOTH
SUBGRAPH
CONTAINS
NOT_CONTAINS
STARTS
STARTS_WITH
NOT_STARTS_WITH
ENDS
ENDS_WITH
NOT_ENDS_WITH
IS_NULL
IS_NOT_NULL
IS_EMPTY
IS_NOT_EMPTY
UNWIND
SKIP
OPTIONAL
THEN
ELSE
END
GROUPS
ZONE
ZONES
INTO
LISTENER
ELASTICSEARCH
FULLTEXT
AUTO
FUZZY
PREFIX
REGEXP
WILDCARD
TEXT
SEARCH
CLIENTS
SIGN
SERVICE
TEXT_SEARCH
RESET
PLAN
COMMENT
SESSIONS
SESSION
SAMPLE
QUERIES
QUERY
KILL
TOP
TRUE
FALSE
```

Last update: August 24, 2021

4.1.6 nGQL style guide

nGQL does not have strict formatting requirements, but creating nGQL statements according to an appropriate and uniform style can improve readability and avoid ambiguity. Using the same nGQL style in the same organization or project helps reduce maintenance costs and avoid problems caused by format confusion or misunderstanding. This topic will provide a style guide for writing nGQL statements.

Compatibility

The styles of nGQL and [Cypher Style Guide](#) are different.

Newline

1. Start a new line to write a clause.

Not recommended:

```
GO FROM "player100" OVER follow REVERSELY YIELD follow._dst AS id;
```

Recommended:

```
GO FROM "player100" \
OVER follow REVERSELY \
YIELD follow._dst AS id;
```

2. Start a new line to write different statements in a composite statement.

Not recommended:

```
GO FROM "player100" 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;
```

Recommended:

```
GO FROM "player100" \
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;
```

3. If the clause exceeds 80 characters, start a new line at the appropriate place.

Not recommended:

```
MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
WHERE (v2.name STARTS WITH "Y" AND v2.age > 35 AND v2.age < v.age) OR (v2.name STARTS WITH "T" AND v2.age < 45 AND v2.age > v.age) \
RETURN v2;
```

Recommended:

```
MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
WHERE (v2.name STARTS WITH "Y" AND v2.age > 35 AND v2.age < v.age) \
OR (v2.name STARTS WITH "T" AND v2.age < 45 AND v2.age > v.age) \
RETURN v2;
```

Note

If needed, you can also start a new line for better understanding, even if the clause does not exceed 80 characters.

Identifier naming

In nGQL statements, characters other than keywords, punctuation marks, and blanks are all identifiers. Recommended methods to name the identifiers are as follows.

1. Use singular nouns to name tags, and use the base form of verbs or verb phrases to form Edge types.

Not recommended:

```
MATCH p=(v:players)-[e:are_following]-(v2) \
RETURN nodes(p);
```

Recommended:

```
MATCH p=(v:player)-[e:follow]-(v2) \
RETURN nodes(p);
```

2. Use the snake case to name identifiers, and connect words with underscores (_) with all the letters lowercase.

Not recommended:

```
MATCH (v:basketballTeam) \
RETURN v;
```

Recommended:

```
MATCH (v:basketball_team) \
RETURN v;
```

Pattern

1. Start a new line on the right side of the arrow indicating an edge when writing patterns.

Not recommended:

```
MATCH (v:player{name: "Tim Duncan", age: 42}) \
-[e:follow]->()-[e:serve]->()<--(v3) \
RETURN v, e, v2;
```

Recommended:

```
MATCH (v:player{name: "Tim Duncan", age: 42})-[e:follow]-> \
()-[e:serve]->()<--(v3) \
RETURN v, e, v2;
```

2. Anonymize the vertices and edges that do not need to be queried.

Not recommended:

```
MATCH (v:player)-(e:follow)->(v2) \
RETURN v;
```

Recommended:

```
MATCH (v:player)-(:follow)->() \
RETURN v;
```

3. Place named vertices in front of anonymous vertices.

Not recommended:

```
MATCH ()-(:follow)->(v) \
RETURN v;
```

Recommended:

```
MATCH (v)<-(:follow)-() \
RETURN v;
```

String

The strings should be surrounded by double quotes.

Not recommended:

```
RETURN 'Hello Nebula!';
```

Recommended:

```
RETURN "Hello Nebula!\\"123\"";
```

Note

When single or double quotes need to be nested in a string, use a backslash () to escape. For example:

```
RETURN \"Nebula Graph is amazing,\" the user says.;"
```

Statement termination

1. End the nGQL statements with an English semicolon (;).

Not recommended:

```
FETCH PROP ON player "player100"
```

Recommended:

```
FETCH PROP ON player "player100";
```

2. Use a pipe (|) to separate a composite statement, and end the statement with an English semicolon at the end of the last line.

Using an English semicolon before a pipe will cause the statement to fail.

Not supported:

```
GO FROM "player100" \
OVER follow \
YIELD follow._dst AS id; | \
GO FROM $-.id \
OVER serve \
YIELD $$team.name AS Team, $^player.name AS Player;
```

Supported:

```
GO FROM "player100" \
OVER follow \
YIELD follow._dst AS id | \
GO FROM $-.id \
OVER serve \
YIELD $$team.name AS Team, $^player.name AS Player;
```

3. In a composite statement that contains user-defined variables, use an English semicolon to end the statements that define the variables. If you do not follow the rules to add a semicolon or use a pipe to end the composite statement, the execution will fail.

Not supported:

```
$var = GO FROM "player100" \
OVER follow \
YIELD follow._dst AS id \
GO FROM $var.id \
OVER serve \
YIELD $$team.name AS Team, $^player.name AS Player;
```

Not supported:

```
$var = GO FROM "player100" \
OVER follow \
YIELD follow._dst AS id | \
GO FROM $var.id \
OVER serve \
YIELD $$team.name AS Team, $^player.name AS Player;
```

Supported:

```
$var = GO FROM "player100" \
OVER follow \
YIELD follow._dst AS id; \
GO FROM $var.id \
OVER serve \
YIELD $$ .team.name AS Team, $^.player.name AS Player;
```

Last update: July 16, 2021

4.2 Data types

4.2.1 Numeric types

nGQL supports both integer and floating-point number.

Integer

Signed 64-bit integer (INT64), 32-bit integer (INT32), 16-bit integer (INT16), and 8-bit integer (INT8) are supported.

Type	Declared keywords	Range
INT64	INT64 or INT	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
INT32	INT32	-2,147,483,648 ~ 2,147,483,647
INT16	INT16	-32,768 ~ 32,767
INT8	INT8	-128 ~ 127

Floating-point number

Both single-precision floating-point format (FLOAT) and double-precision floating-point format (DOUBLE) are supported.

Type	Declared keywords	Range	Precision
FLOAT	FLOAT	3.4E +/- 38	6~7 bits
DOUBLE	DOUBLE	1.7E +/- 308	15~16 bits

Scientific notation is also supported, such as `1e2`, `1.1e2`, `.3e4`, `1.e4`, and `-1234E-10`.

Note

The data type of DECIMAL in MySQL is not supported.

Reading and writing of data values

When writing and reading different types of data, nGQL complies with the following rules:

Data type	Set as VID	Set as property	Resulted data type
INT64	Supported	Supported	INT64
INT32	Not supported	Supported	INT64
INT16	Not supported	Supported	INT64
INT8	Not supported	Supported	INT64
FLOAT	Not supported	Supported	DOUBLE
DOUBLE	Not supported	Supported	DOUBLE

For example, nGQL does not support setting VID as INT8, but supports setting a certain property type of TAG or Edge type as INT8. When using the nGQL statement to read the property of INT8, the resulted type is INT64.

Multiple formats are supported:

- Decimal, such as `123456`.
- Hexadecimal, such as `0x1e240`.
- Octal, such as `0361100`.

However, Nebula Graph will parse the written non-decimal value into a decimal value and save it. The value read is decimal.

For example, the type of the property `score` is `INT`. The value of `0xb` is assigned to it through the `INSERT` statement. If querying the property value with statements such as `FETCH`, you will get the result `11`, which is the decimal result of the hexadecimal `0xb`.

Last update: August 23, 2021

4.2.2 Boolean

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

nGQL supports using boolean in the following ways:

- Define the data type of the property value as a boolean.
 - Use boolean as judgment conditions in the `WHERE` clause.
-

Last update: August 23, 2021

4.2.3 String

Fixed-length strings and variable-length strings are supported.

String types

The string type is declared with the keywords of:

- `STRING` : Variable-length strings.
- `FIXED_STRING(<length>)` : Fixed-length strings. `<length>` is the length of the string, such as `FIXED_STRING(32)`.

A 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. For example, `"Hello, Cooper"` or `'Hello, Cooper'`.

String types

Nebula Graph supports using string types in the following ways:

- Define the data type of `VID` as a fixed-length string.
- Set the variable-length string as the Schema name, including the names of the graph space, tag, edge type, and property.
- Define the data type of the property as a fixed-length or variable-length string.

For example:

- Define the data type of the property as a fixed-length string
- ```
nebula> CREATE TAG t1 (p1 FIXED_STRING(10));
```
- Define the data type of the property as a variable-length string
- ```
nebula> CREATE TAG t2 (p2 STRING);
```

When writing a fixed-length string, if the string you try to write exceeds the length limit, Nebula Graph will truncate the string and store the part that meets the length limit. For example, when the VID type of a graph space is `FIXED_STRING(8)`, if you try to set `A_string_with_27_characters` as a VID and use the nGQL command to query the VID, only `A_string` will be returned.

Escape Characters

Line breaks are not allowed in a string. Escape characters are supported within strings, for example:

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

OpenCypher Compatibility

There are some tiny differences between openCypher and Cypher, as well as nGQL. The following is what openCypher requires. Single quotes cannot be converted to double quotes.

```
# File: Literals.feature
Feature: Literals

Background:
  Given any graph
Scenario: Return a single-quoted string
  When executing query:
  """
    RETURN '' AS literal
  """
  Then the result should be, in any order:
  | literal |
  | '' |    # Note: it should return single-quotes as openCypher required.
  And no side effects
```

While Cypher accepts both single quotes and double quotes as the return results. nGQL follows the Cypher way.

```
nebula > YIELD '' AS quote1, "" AS quote2, """ AS quote3, """ AS quote4
+-----+-----+-----+
| quote1 | quote2 | quote3 | quote4 |
+-----+-----+-----+
| ""     | ""     | """   | """   |
+-----+-----+-----+
```

Last update: August 23, 2021

4.2.4 Date and time types

This topic will describe the `DATE`, `TIME`, `DATETIME`, and `TIMESTAMP` types.

While inserting time-type property values, except for `TIMESTAMP`, Nebula Graph transforms them to a UTC time according to the time zone specified with the `timezone_name` parameter in the [configuration files](#). The time-type values returned by nGQL queries are all UTC time.

Note

To change the time zone, modify the `timezone_name` value in the configuration files of all Nebula Graph services.

- `date()`, `time()`, `datetime()`, and `timestamp()` all accept empty parameters to return the current date, time, and datetime.
- `date()`, `time()`, and `datetime()` all accept the property name to return a specific property value of itself. For example, `date().month` returns the current month, while `time("02:59:40").minute` returns the minutes of the importing time.

OpenCypher Compatibility

In nGQL:

- Year, month, day, hour, minute, and second are supported, while the millisecond is not supported.
- `localdatetime()` and `duration()` are not supported.
- Most string time formats are not supported. The only exception is `YYYY-MM-DDThh:mm:ss`.

DATE

The `DATE` type is used for values with a date part but no time part. Nebula Graph retrieves and displays `DATE` values in the `YYYY-MM-DD` format. The supported range is `-32768-01-01` to `32767-12-31`.

The properties of `date()` include `year`, `month`, and `day`.

TIME

The `TIME` type is used for values with a time part but no date part. Nebula Graph retrieves and displays `TIME` values in `hh:mm:ss.msmsmsususus` format. The supported range is `00:00:00.000000` to `23:59:59.999999`.

The properties of `time()` include `hour`, `minute`, and `second`.

DATETIME

The `DATETIME` type is used for values that contain both date and time parts. Nebula Graph retrieves and displays `DATETIME` values in `YYYY-MM-DDThh:mm:ss.msmsmsususus` format. The supported range is `-32768-01-01T00:00:00.000000` to `32767-12-31T23:59:59.999999`.

The properties of `datetime()` include `year`, `month`, `day`, `hour`, `minute`, and `second`.

TIMESTAMP

The `TIMESTAMP` data type is used for values that contain both date and time parts. It has a range of `1970-01-01T00:00:01` UTC to `2262-04-11T23:47:16` UTC.

`TIMESTAMP` has the following features:

- Stored and displayed in the form of a timestamp, such as `1615974839`, which means `2021-03-17T17:53:59`.
- Supported `TIMESTAMP` querying methods: `timestamp` and `timestamp()` function.
- Supported `TIMESTAMP` inserting methods: `timestamp`, `timestamp()` function, and `now()` function.
- `timestamp()` function accepts empty parameters to get the timestamp of the current time zone and also accepts a string type parameter.

```
# Return the current time.
nebula> return timestamp();
+-----+
| timestamp() |
+-----+
| 1625469277 |
+-----+

# Return the specified time.
nebula> return timestamp("2021-07-05T06:18:43.984000");
+-----+
| timestamp("2021-07-05T06:18:43.984000") |
+-----+
| 1625465923 |
+-----+
```

- The underlying storage data type is **int64**.

Examples

1. Create a tag named `date1` with three properties: `DATE`, `TIME`, and `DATETIME`.

```
nebula> CREATE TAG date1(p1 date, p2 time, p3 datetime);
```

2. Insert a vertex named `test1`.

```
nebula> INSERT VERTEX date1(p1, p2, p3) VALUES "test1":(date("2021-03-17"), time("17:53:59"), datetime("2021-03-17T17:53:59"));
```

3. Return the month of the property `p1` on `test1`.

```
nebula> CREATE TAG INDEX date1_index ON date1(p1);
nebula> REBUILD TAG INDEX date1_index;
nebula> MATCH (v:date1) RETURN v.p1.month;
+-----+
| v.p1.month |
+-----+
| 3          |
+-----+
```

4. Create a tag named `school` with the property of `TIMESTAMP`.

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

5. Insert a vertex named `DUT` with a found-time timestamp of `"1988-03-01T08:00:00"`.

```
# Insert as a timestamp. The corresponding timestamp of 1988-03-01T08:00:00 is 573177600, or 573206400 UTC.
nebula> INSERT VERTEX school(name, found_time) VALUES "DUT":("DUT", 573206400);

# Insert in the form of date and time.
nebula> INSERT VERTEX school(name, found_time) VALUES "DUT":("DUT", timestamp("1988-03-01T08:00:00"));
```

6. Insert a vertex named `dut` and store time with `now()` or `timestamp()` functions.

```
# Use now() function to store time
nebula> INSERT VERTEX school(name, found_time) VALUES "dut":("dut", now());

# Use timestamp() function to store time
nebula> INSERT VERTEX school(name, found_time) VALUES "dut":("dut", timestamp());
```

You can also use `WITH` statement to set a specific date and time. For example:

```
nebula> WITH time({hour: 12, minute: 31, second: 14}) AS d RETURN d;
+-----+
| d   |
+-----+
```

```
| 12:31:14.000 |
+-----+
nebula> WITH date({year: 1984, month: 10, day: 11}) AS x RETURN x + 1;
+-----+
| x   |
+-----+
| 1984-10-12 |
+-----+
```

Last update: August 23, 2021

4.2.5 NULL

You can set the properties for vertices or edges to `NULL`. Also, you can set the `NOT NULL` constraint to make sure that the property values are `NOT NULL`. If not specified, the property is set to `NULL` by default.

Logical operations with NULL

Here is the truth table for `AND`, `OR`, `XOR`, and `NOT`.

a	b	a AND b	a OR b	a XOR b	NOT a
false	false	false	false	false	true
false	null	false	null	null	true
false	true	false	true	true	true
true	false	false	true	true	false
true	null	null	true	null	false
true	true	true	true	false	false
null	false	false	null	null	null
null	null	null	null	null	null
null	true	null	true	null	null

OpenCypher compatibility

The comparisons and operations about `NULL` are different from openCypher. There may be changes later.

COMPARISONS WITH NULL

The comparison operations with `NULL` are incompatible with openCypher.

OPERATIONS AND RETURN WITH NULL

The `NULL` operations and `RETURN` with `NULL` are incompatible with openCypher.

Examples

USE NOT NULL

Create a tag named `player`. Specify the property `name` as `NOT NULL`.

```
nebula> CREATE TAG player(name string NOT NULL, age int);
```

Use `SHOW` to create tag statements. The property `name` is `NOT NULL`. The property `age` is `NULL` by default.

```
nebula> SHOW CREATE TAG player;
+-----+-----+
| Tag   | Create Tag          |
+-----+-----+
| "student" | "CREATE TAG `player` (          |
|           |   `name` string NOT NULL,          |
|           |   `age` int64 NULL              |
|           | ) ttl_duration = 0, ttl_col = ""  |
+-----+-----+
```

Insert the vertex `Kobe`. The property `age` can be `NULL`.

```
nebula> INSERT VERTEX player(name, age) VALUES "Kobe":("Kobe",null);
```

USE NOT NULL AND SET THE DEFAULT

Create a tag named `player`. Specify the property `age` as `NOT NULL`. The default value is `18`.

```
nebula> CREATE TAG player(name string, age int NOT NULL DEFAULT 18);
```

Insert the vertex `Kobe`. Specify the property `name` only.

```
nebula> INSERT VERTEX player(name) VALUES "Kobe"("Kobe");
```

Query the vertex `Kobe`. The property `age` is `18` by default.

```
nebula> FETCH PROP ON player "Kobe"
+-----+
| vertices
+-----+
| ("Kobe" :player{age: 18, name: "Kobe"}) |
+-----+
```

Last update: August 23, 2021

4.2.6 Lists

The list is a composite data type. A list is a sequence of values. Individual elements in a list can be accessed by their positions.

A list starts with a left square bracket `[` and ends with a right square bracket `]`. A list contains zero, one, or more expressions. List elements are separated from each other with commas `(,)`. Whitespace around elements is ignored in the list, thus line breaks, tab stops, and blanks can be used for formatting.

List operations

You can use the preset [list function](#) to operate the list, or use the index to filter the elements in the list.

INDEX SYNTAX

```
[M]
[M..N]
[M..]
[..N]
```

The index of nGQL supports queries from front to back, starting from 0. 0 means the first element, 1 means the second element, and so on. It also supports queries from back to front, starting from -1. -1 means the last element, -2 means the penultimate element, and so on.

- `[M]:` represents the element whose index is `M`.
- `[M..N]:` represents the elements whose indexes are greater or equal to `M` but smaller than `N`. Return empty when `N` is 0.
- `[M..]:` represents the elements whose indexes are greater or equal to `M`.
- `[..N]:` represents the elements whose indexes are smaller than `N`. Return empty when `N` is 0.

Note

- Return empty if the index is out of bounds, while return normally if the index is within the bound.
- Return empty if `M ≥ N`.
- When querying a single element, if `M` is `null` return `BAD_TYPE`. When conducting a range query, if `M` or `N` is `null`, return `null`.

Examples

```
# The following query returns the list [1,2,3].
nebula> RETURN [1, 2, 3] AS List;
+-----+
| List      |
+-----+
| [1, 2, 3] |
+-----+


# The following query returns the element whose index is 3 in the list [1,2,3,4,5]. In a list, the index starts from 0, and thus the return element is 4.
nebula> RETURN range(1,5)[3];
+-----+
| range(1,5)[3] |
+-----+
| 4           |
+-----+


# The following query returns the element whose index is -2 in the list [1,2,3,4,5]. The index of the last element in a list is -1, and thus the return element is 4.
nebula> RETURN range(1,5)[-2];
+-----+
| range(1,5)[-2] |
+-----+
| 4           |
+-----+


# The following query returns the elements whose indexes are from 0 to 3 (not including 3) in the list [1,2,3,4,5].
nebula> RETURN range(1,5)[0..3];
+-----+
| range(1,5)[0..3] |
+-----+
| [1, 2, 3]       |
+-----+
```

```

# The following query returns the elements whose indexes are greater than 2 in the list [1,2,3,4,5].
nebula> RETURN range(1,5)[3..] AS a;
+-----+
| a    |
+-----+
| [4, 5] |
+-----+

# The following query returns the elements whose indexes are smaller than 3.
nebula> WITH [1, 2, 3, 4, 5] AS list \
    RETURN list[..3] AS r;
+-----+
| r    |
+-----+
| [1, 2, 3] |
+-----+

# The following query filters the elements whose indexes are greater than 2 in the list [1,2,3,4,5], calculate them respectively, and returns them.
nebula> RETURN [n IN range(1,5) WHERE n > 2 | n + 10] AS a;
+-----+
| a    |
+-----+
| [13, 14, 15] |
+-----+

# The following query returns the elements from the first to the penultimate (inclusive) in the list [1, 2, 3].
nebula> YIELD [1, 2, 3][0..-1] AS a;
+-----+
| a    |
+-----+
| [1, 2] |
+-----+

# The following query returns the elements from the first (exclusive) to the third backward in the list [1, 2, 3, 4, 5].
nebula> YIELD [1, 2, 3, 4, 5][-3..-1] AS a;
+-----+
| a    |
+-----+
| [3, 4] |
+-----+

# The following query sets the variables and returns the elements whose indexes are 1 and 2.
nebula> $var = YIELD 1 AS f, 3 AS t; \
    YIELD [1, 2, 3][$var.f..$var.t] AS a;
+-----+
| a    |
+-----+
| [2, 3] |
+-----+

# The following query returns empty because the index is out of bound. It will return normally when the index is within the bound.
nebula> RETURN [1, 2, 3, 4, 5] [0..10] AS a;
+-----+
| a    |
+-----+
| [1, 2, 3, 4, 5] |
+-----+

nebula> RETURN [1, 2, 3] [-5..5] AS a;
+-----+
| a    |
+-----+
| [1, 2, 3] |
+-----+

# The following query returns empty because there is a [0..0].
nebula> RETURN [1, 2, 3, 4, 5] [0..0] AS a;
+-----+
| a    |
+-----+
| []  |
+-----+

# The following query returns empty because of M ≥ N.
nebula> RETURN [1, 2, 3, 4, 5] [3..1] AS a;
+-----+
| a    |
+-----+
| []  |
+-----+

# When conduct a range query, if `M` or `N` is null, return `null`.
nebula> WITH [1,2,3] AS list \
    RETURN list[0..null] as a;
+-----+
| a    |
+-----+
| __NULL__ |
+-----+

# The following query calculates the elements in the list [1,2,3,4,5] respectively and returns them without the list head.
nebula> RETURN tail([n IN range(1, 5) | 2 * n - 10]) AS a;
+-----+
| a    |
+-----+

```

```

+-----+
| [-6, -4, -2, 0] |
+-----+
# The following query takes the elements in the list [1,2,3] as true and return.
nebula> RETURN [n IN range(1, 3) WHERE true | n] AS r;
+-----+
| r   |
+-----+
| [1, 2, 3] |
+-----+
# The following query returns the length of the list [1,2,3].
nebula> RETURN size([1,2,3]);
+-----+
| size([1,2,3]) |
+-----+
| 3             |
+-----+
# The following query calculates the elements in the list [92,90] and runs a conditional judgment in a where clause.
nebula> GO FROM "player100" OVER follow WHERE follow.degree NOT IN [x IN [92, 90] | x + $$player.age] \
    YIELD follow._dst AS id, follow.degree AS degree;
+-----+-----+
| id      | degree |
+-----+-----+
| "player101" | 95   |
+-----+-----+
| "player102" | 90   |
+-----+-----+
# The following query takes the query result of the MATCH statement as the elements in a list. Then it calculates and returns them.
nebula> MATCH p = (n:player{name:"Tim Duncan"})-[:follow]->(m) \
    RETURN [n IN nodes(p) | n.age + 100] AS r;
+-----+
| r   |
+-----+
| [142, 136] |
+-----+
| [142, 133] |
+-----+

```

OpenCypher compatibility

- In openCypher, return `null` when querying a single out-of-bound element. However, in nGQL, return `OUT_OF_RANGE` when querying a single out-of-bound element.

```

nebula> RETURN range(0,5)[-12];
+-----+
| range(0,5)[-12] |
+-----+
| OUT_OF_RANGE   |
+-----+

```

- A composite data type (i.e., set, map, and list) **CAN NOT** be stored as properties for vertices or edges.
 - + It is recommended to modify the graph modeling method. The composite data type should be modeled as an adjacent edge of a vertex, rather than its property. Each adjacent edge can be dynamically added or deleted. The rank values of the adjacent edges can be used for sequencing.
- Patterns are not supported in the list. For example, `[(src)-[]->(m) | m.name]`.

Last update: July 14, 2021

4.2.7 Sets

The set is a composite data type.

OpenCypher compatibility

A set is not a data type in openCypher. The behavior of a set in nGQL is not determined yet.

Last update: July 14, 2021

4.2.8 Maps

The map is a composite data type. Maps are unordered collections of key-value pairs. In maps, the key is a string. The value can have any data type. You can get the map element by using `map['key']`.

Literal maps

```
nebula> YIELD {key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]}\n+-----+\n| {key:Value,listKey:[{inner:Map1},{inner:Map2}]} |\n+-----+\n| {key: "Value", listKey: [{inner: "Map1"}, {inner: "Map2"}]} |\n+-----+
```

OpenCypher compatibility

- A composite data type (i.e. set, map, and list) **CANNOT** be stored as properties of vertices or edges.
- Map projection is not supported.

Last update: July 14, 2021

4.2.9 Type Conversion/Type coercions

Converting an expression of a given type to another type is known as type conversion.

Legacy version compatibility

- nGQL 1.0 adopts the C-style of type conversion (implicitly or explicitly): `(type_name)expression`. For example, the results of `YIELD (int)(TRUE)` is `1`. But it is error-prone to users who are not familiar with the C language.
- nGQL 2.0 chooses the openCypher way of type coercions.

Type coercions functions

Function	Description
<code>toBoolean()</code>	Converts a string value to a boolean value.
<code>toFloat()</code>	Converts an integer or string value to a floating point number.
<code>toInteger()</code>	Converts a floating point or string value to an integer value.
<code>type()</code>	Returns the string representation of the relationship type.

Examples

```
nebula> UNWIND [true, false, 'true', 'false', NULL] AS b RETURN toBoolean(b) AS b;
+-----+
| b   |
+-----+
| true |
+-----+
| false |
+-----+
| true |
+-----+
| false |
+-----+
| __NULL__ |
+-----+


nebula> RETURN toFloat(1), toFloat('1.3'), toFloat('1e3'), toFloat('not a number');
+-----+-----+-----+-----+
| toFloat(1) | toFloat("1.3") | toFloat("1e3") | toFloat("not a number") |
+-----+-----+-----+-----+
| 1.0       | 1.3          | 1000.0       | __NULL__      |
+-----+-----+-----+-----+


nebula> RETURN toInteger(1), toInteger('1'), toInteger('1e3'), toInteger('not a number');
+-----+-----+-----+-----+
| toInteger(1) | toInteger("1") | toInteger("1e3") | toInteger("not a number") |
+-----+-----+-----+-----+
| 1          | 1            | 1000         | __NULL__      |
+-----+-----+-----+-----+


nebula> MATCH (a:player)-[e]-() RETURN type(e);
+-----+
| type(e) |
+-----+
| "follow" |
+-----+
| "follow" |


nebula> MATCH (a:player {name: "Tim Duncan"}) WHERE toInteger(id(a)) == 100 RETURN a;
+-----+
| a   |
+-----+
| {"100": player{age: 42, name: "Tim Duncan"} } |
+-----+


nebula> MATCH (n:player) WITH n LIMIT toInteger(ceil(1.8)) RETURN count(*) AS count;
+-----+
| count |
+-----+
| 2     |
+-----+
```

Last update: July 14, 2021

4.3 Variables and composite queries

4.3.1 Composite queries (clause structure)

Composite queries put data from different queries together. They then use filters, group-bys, or sorting before returning the combined return results.

Nebula Graph supports three methods to run composite queries (or sub-queries):

- (openCypher) Clauses are chained together, and they feed intermediate result sets between each other.
- (Native nGQL) More than one query can be batched together, separated by semicolons (;). The result of the last query is returned as the result of the batch.
- (Native nGQL) Queries can be piped together by using the pipe (|). The result of the previous query can be used as the input of the next query.

OpenCypher compatibility

In a composite query, **do not** put together openCypher and native nGQL clauses in one statement. For example, this statement is undefined: `MATCH ... | GO ... | YIELD ...`.

- If you are in the openCypher way (`MATCH`, `RETURN`, `WITH`, etc), do not introduce any pipe or semicolons to combine the sub-clauses.
- If you are in the native nGQL way (`FETCH`, `GO`, `LOOKUP`, etc), you must use pipe or semicolons to combine the sub-clauses.

Composite queries are not transactional queries (as in SQL/Cypher)

For example, a query is composed of three sub-queries: `A B C`, `A | B | C` or `A; B; C`. In that A is a read operation, B is a computation operation, and C is a write operation. If any part fails in the execution, the whole result will be undefined. There is no rollback. What is written depends on the query executor.

Note

OpenCypher has no requirement of `transaction`.

Examples

- OpenCypher compatibility statement

```
# Connect multiple queries with clauses.
nebula> MATCH p=(v:player{name:"Tim Duncan"})--() \
    WITH nodes(p) AS n \
```

```
UNWIND n AS n1 \
RETURN DISTINCT n1;
```

- Native nGQL (Semicolon queries)

```
# Only return edges.
nebula> SHOW TAGS; SHOW EDGES;

# Insert multiple vertices.
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42); \
    INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36); \
    INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
```

- Native nGQL (Pipe queries)

```
# Connect multiple queries with pipes.
nebula> GO FROM "player100" OVER follow YIELD follow._dst AS id | \
    GO FROM $.id OVER serve YIELD $$team.name AS Team, \
    $^.player.name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| Nuggets | Tony Parker |
```

Last update: July 14, 2021

4.3.2 User-defined variables

User-defined variables allow passing the result of one statement to another.

OpenCypher compatibility

In openCypher, when you refer to the vertex, edge, or path of a variable, you need to name it first. For example:

```
nebula> MATCH (v:player{name:"Tim Duncan"}) RETURN v;
+-----+
| v
+-----+
| ("player100" :player{name: "Tim Duncan", age: 42}) |
+-----+
```

The user-defined variable in the preceding query is `v`.

Native nGQL

User-defined variables are written as `$var_name`. The `var_name` consists of letters, numbers, or underline characters. Any other characters are not permitted.

The user-defined variables are valid only at the current execution. When the execution ends, the user-defined variables will be automatically expired. The user-defined variables in one statement **CANNOT** be used in either other clients or other executions.

You can use user-defined variables in composite queries. Details about composite queries, see [Composite queries](#).

Note

User-defined variables are case-sensitive.

Example

```
nebula> $var = GO FROM "player100" OVER follow YIELD follow._dst AS id; \
GO FROM $var.id OVER serve YIELD $$.team.name AS Team, \
$^.player.name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| Nuggets | Tony Parker |
+-----+-----+
```

Last update: July 14, 2021

4.3.3 Property reference

You can refer to the properties of a vertex or an edge in `WHERE` and `YIELD` syntax.

Note

This function applies to native nGQL only.

Property reference for vertex

FOR SOURCE VERTEX

```
$^.<tag_name>.<prop_name>
```

Parameter	Description
<code>\$^</code>	is used to get the property of the source vertex.
<code>tag_name</code>	is the tag name of the vertex.
<code>prop_name</code>	specifies the property name.

FOR DESTINATION VERTEX

```
$.<tag_name>.<prop_name>
```

Parameter	Description
<code>\$\$</code>	is used to get the property of the destination vertex.
<code>tag_name</code>	is the tag name of the vertex.
<code>prop_name</code>	specifies the property name.

Property reference for edge

FOR USER-DEFINED EDGE PROPERTY

```
<edge_type>.<prop_name>
```

Parameter	Description
<code>edge_type</code>	is the edge type of the edge.
<code>prop_name</code>	specifies the property name of the edge type.

FOR BUILT-IN PROPERTIES

Apart from the user-defined edge property, there are four built-in properties in each edge:

Parameter	Description
<code>_src</code>	source vertex ID of the edge
<code>_dst</code>	destination vertex ID of the edge
<code>_type</code>	edge type
<code>_rank</code>	the rank value for the edge

Examples

```

# The following query returns the `name` property of the source vertex and the `age` property of the destination vertex.
nebula> GO FROM "player100" OVER follow YIELD $^.player.name AS startName, $$^.player.age AS endAge;
+-----+-----+
| startName | endAge |
+-----+-----+
| "Tim Duncan" | 36 |
+-----+-----+
| "Tim Duncan" | 33 |
+-----+-----+

# The following query returns the `degree` property of the edge.
nebula> GO FROM "player100" OVER follow YIELD follow.degree;
+-----+
| follow.degree |
+-----+
| 95 |
+-----+
| 90 |
+-----+

# The following query returns the source vertex, the destination vertex, edge type, and edge rank value of the `follow` edge.
nebula> GO FROM "player100" OVER follow YIELD follow._src, follow._dst, follow._type, follow._rank;
+-----+-----+-----+-----+
| follow._src | follow._dst | follow._type | follow._rank |
+-----+-----+-----+-----+
| "player100" | "player101" | 136 | 0 |
+-----+-----+-----+-----+
| "player100" | "player102" | 136 | 0 |
+-----+-----+-----+

```

Last update: July 14, 2021

4.4 Operators

4.4.1 Comparison operators

Nebula Graph supports the following comparison operators.

Name	Description
=	Assigns a value
+	Addition operator
-	Minus operator
*	Multiplication operator
/	Division operator
==	Equal operator
!=, <>	Not equal operator
>	Greater than operator
>=	Greater than or equal operator
<	Less than operator
<=	Less than or equal operator
%	Modulo operator
-	Changes the sign of the argument
IS NULL	NULL check
IS NOT NULL	Not NULL check
IS EMPTY	EMPTY check
IS NOT EMPTY	Not EMPTY check

The result of the comparison operation is `true` or `false`.

Note

- Comparability between values of different types is often undefined. The result could be `NULL` or others.
- `EMPTY` is currently used only for checking and does not support functions or operations such as `GROUP BY`, `count()`, `sum()`, `max()`, `hash()`, `collect()`, `+` or `*`.

OpenCypher compatibility

- The comparison operation of `NULL` is different from openCypher. The behavior may also change. `IS [NOT] NULL` is often used with `OPTIONAL MATCH` in openCypher. But `OPTIONAL MATCH` is not supported in nGQL.
- openCypher does not have `EMPTY`. Thus `EMPTY` is not supported in MATCH statements.

Examples

==

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

Note

The equal operator is `==` in nGQL, while in openCypher it is `=`.

```
nebula> RETURN 'A' == 'a', toUpper('A') == toUpper('a'), toLower('A') == toLower('a');
+-----+-----+-----+
| ("A"=="a") | (toUpper("A")==toUpper("a")) | (toLower("A")==toLower("a")) |
+-----+-----+-----+
| false      | true           | true           |
+-----+-----+-----+
```

```
nebula> RETURN '2' == 2, toInteger('2') == 2;
+-----+-----+
| ("2"==2) | (toInteger("2")==2) |
+-----+-----+
| false      | true           |
+-----+-----+
```

>

```
nebula> RETURN 3 > 2;
+-----+
| (3>2) |
+-----+
| true  |
+-----+
```

```
nebula> WITH 4 AS one, 3 AS two \
    RETURN one > two AS result;
+-----+
| result |
+-----+
| true  |
+-----+
```

>=

```
nebula> RETURN 2 >= "2", 2 >= 2;
+-----+-----+
| (2>="2") | (2>=2) |
+-----+-----+
| __NULL__ | true   |
+-----+-----+
```

<

```
nebula> YIELD 2.0 < 1.9;
+-----+
| (2<1.9) |
+-----+
| false   |
+-----+
```

<=

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

!=

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

IS [NOT] NULL

```

nebula> RETURN null IS NULL AS value1, null == null AS value2, null != null AS value3;
+-----+-----+-----+
| value1 | value2 | value3 |
+-----+-----+-----+
| true  | _NULL_ | _NULL_ |
+-----+-----+-----+


nebula> RETURN length(NULL), size(NULL), count(NULL), NULL IS NULL, NULL IS NOT NULL, sin(NULL), NULL + NULL, [1, NULL] IS NULL;
+-----+-----+-----+-----+-----+-----+-----+-----+
| length(NULL) | size(NULL) | count(NULL) | NULL IS NULL | NULL IS NOT NULL | sin(NULL) | (NULL+NULL) | [1,NULL] IS NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
| _NULL_      | _NULL_      | 0          | true        | false       | _NULL_    | _NULL_    | false      |
+-----+-----+-----+-----+-----+-----+-----+-----+


nebula> WITH {name: null} AS map \
    RETURN map.name IS NOT NULL;
+-----+
| map.name IS NOT NULL |
+-----+
| false                |
+-----+


nebula> WITH {name: 'Mats', name2: 'Pontus'} AS map1, \
    {name: null} AS map2, {notName: 0, notName2: null } AS map3 \
    RETURN map1.name IS NULL, map2.name IS NOT NULL, map3.name IS NULL;
+-----+-----+-----+
| map1.name IS NULL | map2.name IS NOT NULL | map3.name IS NULL |
+-----+-----+-----+
| false            | false            | true            |
+-----+-----+-----+


nebula> MATCH (n:player) \
    RETURN n.age IS NULL, n.name IS NOT NULL, n.empty IS NULL;
+-----+-----+-----+
| n.age IS NULL | n.name IS NOT NULL | n.empty IS NULL |
+-----+-----+-----+
| false          | true            | true            |
+-----+-----+-----+
| false          | true            | true            |
+-----+-----+-----+
| false          | true            | true            |
+-----+-----+-----+
...
```

IS [NOT] EMPTY

```
nebula> RETURN null IS EMPTY;
+-----+
| NULL IS EMPTY |
+-----+
| false          |
+-----+  
  
nebula> RETURN "a" IS NOT EMPTY;
+-----+
| "a" IS NOT EMPTY |
+-----+
| true           |
+-----+  
  
nebula> GO FROM "player100" OVER * WHERE $$.player.name IS NOT EMPTY YIELD follow._dst;
+-----+
| follow._dst |
+-----+
| "player125" |
+-----+
| "player101" |
+-----+
```

Last update: July 19, 2021

4.4.2 Boolean operators

Nebula Graph supports the following boolean operators.

Name	Description
AND	Logical AND
NOT	Logical NOT
OR	Logical OR
XOR	Logical XOR

For the precedence of the operators, refer to [Operator Precedence](#).

For the logical operations with `NULL`, refer to [NULL](#).

Legacy version compatibility

- In Nebula Graph 2.0, non-zero numbers cannot be converted to boolean values.

Last update: July 19, 2021

4.4.3 Pipe operators

Multiple queries can be combined using pipe operators in nGQL.

OpenCypher compatibility

Pipe operators apply to native nGQL only.

Syntax

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

- In SQL, sub-queries are nested in the query statements.
- In nGQL, the shell style `PIPE (|)` is introduced into the sub-queries.

Examples

```
nebula> GO FROM "player100" OVER follow \
YIELD follow._dst AS dstid, $$.player.name AS Name | \
GO FROM $-.dstid OVER follow;

+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
```

If there is no `YIELD` clause to define the output, the destination vertex ID is returned by default. If a `YIELD` clause is applied, the output is defined by the `YIELD` clause.

You must define aliases in the `YIELD` clause for the reference operator `$-` to use, just like `$-.dstid` in the preceding example.

Performance tips

In Nebula Graph 2.5.0, pipes will affect the performance. Take `A | B` as an example, the effects are as follows:

1. Pipe operators operate synchronously. That is, the data can enter the pipe clause as a whole after the execution of clause `A` before the pipe operator is completed.
2. Pipe operators need to be serialized and deserialized, which is executed in a single thread.
3. If `A` sends a large amount of data to `|`, the entire query request may be very slow. You can try to split this statement.
 - a. Send `A` from the application,
 - b. Split the return results on the application,
 - c. Send to multiple graphd processes concurrently,
 - d. Every graphd process executes part of B.

This is usually much faster than executing a complete `A | B` with a single graphd process.

Last update: July 19, 2021

4.4.4 Reference operators

NGQL provides reference operators to represent a property in a `WHERE` or `YIELD` clause, or the output of the statement before the pipe operator in a composite query.

OpenCypher compatibility

Reference operators apply to native nGQL only.

Reference operator List

Reference operator	Description
<code>\$^</code>	Refers to a source vertex property. For more information, see Property reference .
<code>\$\$</code>	Refers to a destination vertex property. For more information, see Property reference .
<code>\$-</code>	Refers to the output of the statement before the pipe operator in a composite query. For more information, see Pipe .

Examples

```
# The following example returns the age of the source vertex and the destination vertex.
nebula> GO FROM "player100" OVER follow YIELD $^.player.age AS SrcAge, $$ .player.age AS DestAge;
+-----+-----+
| SrcAge | DestAge |
+-----+-----+
| 42     | 36      |
+-----+-----+
| 42     | 41      |
+-----+-----+


# The following example returns the name and team of the players that player100 follows.
nebula> GO FROM "player100" OVER follow \
    YIELD follow._dst AS id | \
    GO FROM $-.id OVER serve \
    YIELD $^.player.name AS Player, $$ .team.name AS Team;
+-----+-----+
| Player      | Team      |
+-----+-----+
| "Tony Parker" | "Spurs"  |
+-----+-----+
| "Tony Parker" | "Hornets" |
+-----+-----+
| "Manu Ginobili" | "Spurs"  |
+-----+-----+
```

Last update: July 19, 2021

4.4.5 Set operators

This topic will describe the set operators, including `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`. To combine multiple queries, use these set operators.

All set operators have equal precedence. If a nGQL statement contains multiple set operators, Nebula Graph will evaluate them from left to right unless parentheses explicitly specify another order.

OpenCypher compatibility

Set operators apply to native nGQL only.

UNION, UNION DISTINCT, and UNION ALL

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

- Operator `UNION DISTINCT` (or by short `UNION`) returns the union of two sets A and B without duplicated elements.
- Operator `UNION ALL` returns the union of two sets A and B with duplicated elements.
- The `<left>` and `<right>` must have the same number of columns and data types. Different data types are converted according to the [Type Conversion](#).

EXAMPLES

```
# The following statement returns the union of two query results without duplicated elements.
nebula> GO FROM "player102" OVER follow \
    UNION \
    GO FROM "player100" OVER follow;
+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
| "player102" |
+-----+


# The following statement returns the union of two query results with duplicated elements.
nebula> GO FROM "player102" OVER follow \
    UNION ALL \
    GO FROM "player100" OVER follow;
+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
| "player101" |
+-----+
| "player102" |
+-----+


# UNION can also work with the YIELD statement. The DISTINCT keyword will check duplication by all the columns for every line, and remove duplicated lines if every column is the same.
nebula> GO FROM "player102" OVER follow \
    YIELD follow._dst AS id, follow.degree AS Degree, $$player.age AS Age \
    UNION /* DISTINCT */ \
    GO FROM "player100" OVER follow \
    YIELD follow._dst AS id, follow.degree AS Degree, $$player.age AS Age;
+-----+-----+-----+
| id      | Degree | Age  |
+-----+-----+-----+
| "player101" | 75    | 36   |
+-----+-----+-----+
| "player101" | 96    | 36   |
+-----+-----+-----+
| "player102" | 90    | 33   |
+-----+-----+-----+
```

INTERSECT

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

- Operator `INTERSECT` returns the intersection of two sets A and B (denoted by $A \cap B$).
- Similar to `UNION`, the `left` and `right` must have the same number of columns and data types. Different data types are converted according to the [Type Conversion](#).

EXAMPLE

```
nebula> GO FROM "player102" OVER follow \
  YIELD follow._dst AS id, follow.degree AS Degree, $$.player.age AS Age \
  INTERSECT \
  GO FROM "player100" OVER follow \
  YIELD follow._dst AS id, follow.degree AS Degree, $$.player.age AS Age;
Empty set (time spent 2990/3511 us)
```

MINUS

```
<left> MINUS <right>
```

Operator `MINUS` returns the subtraction (or difference) of two sets A and B (denoted by $A - B$). Always pay attention to the order of `left` and `right`. The set $A - B$ consists of elements that are in A but not in B.

EXAMPLE

```
nebula> GO FROM "player100" OVER follow \
  MINUS \
  GO FROM "player102" OVER follow;
+-----+
| follow._dst |
+-----+
| "player102" |
+-----+

nebula> GO FROM "player102" OVER follow \
  MINUS \
  GO FROM "player100" OVER follow;
Empty set (time spent 2243/3259 us)
```

Precedence of the set operators and pipe operators

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

EXAMPLES

```
nebula> GO FROM "player102" OVER follow \
  YIELD follow._dst AS play_dst \
  UNION \
  GO FROM "team200" OVER serve REVERSELY \
  YIELD serve._dst AS play_dst \
  | GO FROM $-.play_dst OVER follow YIELD follow._dst AS play_dst;

+-----+
| play_dst   |
+-----+
| "player101" |
+-----+
| "player102" |
+-----+
```

```
nebula> GO FROM "player102" OVER follow YIELD follow._dst AS play_dst \
UNION \
GO FROM "team200" OVER serve REVERSELY YIELD serve._dst AS play_dst \
| GO FROM $-.play_dst OVER follow YIELD follow._dst AS play_dst;
```

The above query executes the statements in the red bar first and then executes the statement in the green box.

The parentheses can change the execution priority. For example:

```
nebula> (GO FROM "player102" OVER follow \
    YIELD follow._dst AS play_dst \
    UNION \
    GO FROM "team200" 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 statements within the parentheses take precedence. That is, the `UNION` operation will be executed first, and its output will be executed as the input of the next operation with pipes.

Last update: July 19, 2021

4.4.6 String operators

You can use the following string operators for concatenating, querying, and matching.

Name	Description
+	Concatenates strings.
CONTAINS	Performs searchings in strings.
(NOT) IN	Checks whether a value is within a set of values.
(NOT) STARTS WITH	Performs matchings at the beginning of a string.
(NOT) ENDS WITH	Performs matchings at the end of a string.
Regular expressions	Perform string matchings using regular expressions.

Note

All the string searchings or matchings are case-sensitive.

Examples

+

```
nebula> RETURN 'a' + 'b';
+-----+
| (a+b) |
+-----+
| "ab" |
+-----+
nebula> UNWIND 'a' AS a UNWIND 'b' AS b RETURN a + b;
+-----+
| (a+b) |
+-----+
| "ab" |
+-----+
```

CONTAINS

The `CONTAINS` operator requires string types on both left and right sides.

```
nebula> MATCH (s:player)-[e:serve]->(t:team) WHERE id(s) == "player101" \
  AND t.name CONTAINS "ets" RETURN s.name, e.start_year, e.end_year, t.name;
+-----+-----+-----+
| s.name | e.start_year | e.end_year | t.name |
+-----+-----+-----+
| "Tony Parker" | 2018 | 2019 | "Hornets" |
+-----+-----+-----+
nebula> GO FROM "player101" OVER serve WHERE (STRING)serve.start_year CONTAINS "19" AND \
  $^.player.name CONTAINS "ny" \
  YIELD $^.player.name, serve.start_year, serve.end_year, $$.team.name;
+-----+-----+-----+
| $^.player.name | serve.start_year | serve.end_year | $$team.name |
+-----+-----+-----+
| "Tony Parker" | 1999 | 2018 | "Spurs" |
+-----+-----+-----+
nebula> GO FROM "player101" OVER serve WHERE !( $$team.name CONTAINS "ets") \
  YIELD $^.player.name, serve.start_year, serve.end_year, $$team.name;
+-----+-----+-----+
| $^.player.name | serve.start_year | serve.end_year | $$team.name |
+-----+-----+-----+
| "Tony Parker" | 1999 | 2018 | "Spurs" |
+-----+-----+-----+
```

(NOT) IN

```
nebula> RETURN 1 IN [1,2,3], "Yao" NOT IN ["Yi", "Tim", "Kobe"], NULL in ["Yi", "Tim", "Kobe"]
+-----+-----+-----+
| (1 IN [1,2,3]) | ("Yao" NOT IN ["Yi", "Tim", "Kobe"]) | (NULL IN ["Yi", "Tim", "Kobe"]) |
+-----+-----+-----+
```

true	true	false	
+	+	+	+

(NOT) STARTS WITH

```
nebula> RETURN 'apple' STARTS WITH 'app', 'apple' STARTS WITH 'a', 'apple' STARTS WITH toUpper('a')
+-----+-----+-----+
| ("apple" STARTS WITH "app") | ("apple" STARTS WITH "a") | ("apple" STARTS WITH toUpper("a")) |
+-----+-----+-----+
| true | true | false |
+-----+-----+-----+  
  
nebula> RETURN 'apple' STARTS WITH 'b','apple' NOT STARTS WITH 'app'
+-----+-----+
| ("apple" STARTS WITH "b") | ("apple" NOT STARTS WITH "app") |
+-----+-----+
| false | false |
+-----+-----+
```

(NOT) ENDS WITH

```
nebula> RETURN 'apple' ENDS WITH 'app', 'apple' ENDS WITH 'e', 'apple' ENDS WITH 'E', 'apple' ENDS WITH 'b'  
+-----+-----+-----+-----+  
| ("apple" ENDS WITH "app")) | ("apple" ENDS WITH "e") | ("apple" ENDS WITH "E") | ("apple" ENDS WITH "b") |  
+-----+-----+-----+-----+  
| false | true | false | false |  
+-----+-----+-----+-----+
```

REGULAR EXPRESSIONS

Note

Regular expressions cannot work with native nGQL statements (`GO`, `FETCH`, `LOOKUP`, etc.). Use it in openCypher only (`MATCH`, `WHERE`, etc.).

Nebula Graph supports filtering by using regular expressions. The regular expression syntax is inherited from `std::regex`. You can match on regular expressions by using `=~ 'regexp'`. For example:

```
nebula> RETURN "384748.39" =~ "\d+(\.\d{2})?";  
+-----+  
| (384748.39=~\d+(\.\d{2})?) |  
+-----+  
| true |  
+-----+  
  
nebula> MATCH (v:player) WHERE v.name =~ 'Tony.*' RETURN v.name;  
+-----+  
| v.name |  
+-----+  
| "Tony Parker" |  
+-----+
```

Last update: July 19, 2021

4.4.7 List operators

Nebula Graph supports the following list operators:

List operator	Description
+	Concatenates lists.
IN	Checks if an element exists in a list.
[]	Accesses an element(s) in a list using the index operator.

Examples

```
nebula> YIELD [1,2,3,4,5]+[6,7] AS myList;
+-----+
| myList          |
+-----+
| [1, 2, 3, 4, 5, 6, 7] |
+-----+


nebula> RETURN size([NULL, 1, 2]);
+-----+
| size([NULL,1,2]) |
+-----+
| 3               |
+-----+


nebula> RETURN NULL IN [NULL, 1];
+-----+
| (NULL IN [NULL,1]) |
+-----+
| true             |
+-----+


nebula> WITH [2, 3, 4, 5] AS numberlist \
    UNWIND numberlist AS number \
    WITH number \
    WHERE number IN [2, 3, 8] \
    RETURN number;
+-----+
| number |
+-----+
| 2      |
+-----+
| 3      |
+-----+


nebula> WITH ['Anne', 'John', 'Bill', 'Diane', 'Eve'] AS names RETURN names[1] AS result;
+-----+
| result |
+-----+
| "John" |
+-----+
```

Last update: July 19, 2021

4.4.8 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)
- `!`, NOT
- `*`, `/`, `%`
- `-`, `+`
- `==`, `>=`, `>`, `<=`, `<`, `<>`, `!=`
- AND
- OR, XOR
- `=` (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 modify this order and group terms explicitly, use parentheses.

Examples

```
nebula> RETURN 2+3*5;
+-----+
| (2+(3*5)) |
+-----+
| 17         |
+-----+
nebula> RETURN (2+3)*5;
+-----+
| ((2+3)*5) |
+-----+
| 25         |
+-----+
```

OpenCypher compatibility

In openCypher, comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y AND y <= z` in openCypher.

But in nGQL, `x < y <= z` is equivalent to `(x < y) <= z`. The result of `(x < y)` is a boolean. Compare it with an integer `z`, and you will get the final result `NULL`.

Last update: July 19, 2021

4.5 Functions and expressions

4.5.1 Built-in math functions

Function descriptions

Nebula Graph supports the following built-in math functions:

Function	Description
double abs(double x)	Returns the absolute value of the argument.
double floor(double x)	Returns the largest integer value smaller than or equal to the argument. (Rounds down)
double ceil(double x)	Returns the smallest integer greater than or equal to the argument. (Rounds up)
double round(double x)	Returns the integer value nearest to the argument. Returns a number farther away from 0 if the argument is in the middle.
double sqrt(double x)	Returns the square root of the argument.
double cbrt(double x)	Returns the cubic root of the argument.
double hypot(double x, double y)	Returns the hypotenuse of a right-angled triangle.
double pow(double x, double y)	Returns the result of (x^y) .
double exp(double x)	Returns the result of (e^x) .
double exp2(double x)	Returns the result of (2^x) .
double log(double x)	Returns the base-e logarithm of the argument.
double log2(double x)	Returns the base-2 logarithm of the argument.
double log10(double x)	Returns the base-10 logarithm of the argument.
double sin(double x)	Returns the sine of the argument.
double asin(double x)	Returns the inverse sine of the argument.
double cos(double x)	Returns the cosine of the argument.
double acos(double x)	Returns the inverse cosine of the argument.
double tan(double x)	Returns the tangent of the argument.
double atan(double x)	Returns the inverse tangent of the argument.
double rand()	Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. $[0,1]$.
int rand32(int min, int max)	Returns a random 32-bit integer in $[min, max]$. If you set only one argument, it is parsed as <code>max</code> and <code>min</code> is <code>0</code> by default. If you set no argument, the system returns a random signed 32-bit integer.
int rand64(int min, int max)	Returns a random 64-bit integer in $[min, max]$. If you set only one argument, it is parsed as <code>max</code> and <code>min</code> is <code>0</code> by default. If you set no argument, the system returns a random signed 64-bit integer.
collect()	Puts all the collected values into a list.
avg()	Returns the average value of the argument.
count()	Returns the number of records.
max()	Returns the maximum value.
min()	Returns the minimum value.
std()	Returns the population standard deviation.
sum()	Returns the sum value.
bit_and()	Bitwise AND.

Function	Description
bit_or()	Bitwise OR.
bit_xor()	Bitwise XOR.
int size()	Returns the number of elements in a list or a map.
int range(int start, int end, int step)	Returns a list of integers from [start, end] in the specified steps. step is 1 by default.
int sign(double x)	Returns the signum of the given number. If the number is 0, the system returns 0. If the number is negative, the system returns -1. If the number is positive, the system returns 1.
double e()	Returns the base of the natural logarithm, e (2.718281828459045).
double pi()	Returns the mathematical constant pi (3.141592653589793).
double radians()	Converts degrees to radians. radians(180) returns 3.141592653589793 .

Note

If the argument is `NULL`, the output is undefined.

Example

```
# The following clause supports aggregate functions.
nebula> GO FROM "Tim Duncan" OVER like._dst AS dst, $$.player.age AS age \
| GROUP BY $._dst \
YIELD \
$._dst AS dst, \
toInteger((sum($._age)/count($._age)) + avg(distinct $._age+1)+1 AS statistics;
+-----+-----+
| dst | statistics |
+-----+-----+
| "Tony Parker" | 74.0 |
+-----+-----+
| "Manu Ginobili" | 84.0 |
+-----+-----+
Got 2 rows (time spent 4739/5064 us)
```

Last update: July 19, 2021

4.5.2 Built-in string functions

Nebula Graph supports the following built-in string functions:

Note

Like SQL, the position index of nGQL starts from `1`, while in C language it starts from `0`.

Function	Description
<code>int strcasecmp(string a, string b)</code>	Compares string a and b without case sensitivity. When <code>a = b</code> , the return value is 0. When <code>a > b</code> , the return value is greater than 0. When <code>a < b</code> , the return value is less than 0.
<code>string lower(string a)</code>	Returns the argument in lowercase.
<code>string toLower(string a)</code>	The same as <code>lower()</code> .
<code>string upper(string a)</code>	Returns the argument in uppercase.
<code>string toUpper(string a)</code>	The same as <code>upper()</code> .
<code>int length(string a)</code>	Returns the length of the given string in bytes.
<code>string trim(string a)</code>	Removes leading and trailing spaces.
<code>string ltrim(string a)</code>	Removes leading spaces.
<code>string rtrim(string a)</code>	Removes trailing spaces.
<code>string left(string a, int count)</code>	Returns a substring consisting of <code>count</code> characters from the left side of string a. If string a is shorter than <code>count</code> , the system returns string a.
<code>string right(string a, int count)</code>	Returns a substring consisting of <code>count</code> characters from the right side of string a. If string a is shorter than <code>count</code> , the system returns string a.
<code>string lpad(string a, int size, string letters)</code>	Left-pads string a with string <code>letters</code> and returns a substring with the length of <code>size</code> .
<code>string rpad(string a, int size, string letters)</code>	Right-pads string a with string <code>letters</code> and returns a substring with the length of <code>size</code> .
<code>string substr(string a, int pos, int count)</code>	Returns a substring extracting <code>count</code> characters starting from the specified position <code>pos</code> of string a.
<code>string substring(string a, int pos, int count)</code>	The same as <code>substr()</code> .
<code>string reverse(string)</code>	Returns a string in reverse order.
<code>string replace(string a, string b, string c)</code>	Replaces string b in string a with string c.
<code>list split(string a, string b)</code>	Splits string a at string b and returns a list of strings.
<code>string toString()</code>	Takes in any data type and converts it into a string.
<code>int hash()</code>	Takes in any data type and encodes it into a hash value.

Note

If the argument is `NULL`, the return is undefined.

Explanations for the return of `substr()` and `substring()`

- The position index starts from `0`.
- If `pos` is `0`, the whole string is returned.
- If `pos` is greater than the maximum string index, an empty string is returned.
- If `pos` is a negative number, `BAD_DATA` is returned.
- If `count` is omitted, the function returns the substring starting at the position given by `pos` and extending to the end of the string.
- If `count` is `0`, an empty string is returned.
- Using `NULL` as any of the argument of `substr()` will cause [an issue](#).

OpenCypher compatibility

- In openCypher, if `a` is `null`, `null` is returned.
- In openCypher, if `pos` is `0`, the returned substring starts from the first character, and extend to `count` characters.
- In openCypher, if either `pos` or `count` is `null` or a negative integer, an issue is raised.

Last update: July 19, 2021

4.5.3 Built-in date and time functions

Nebula Graph supports the following built-in date and time functions:

Function	Description
int now()	Returns the current date and time of the system time zone.
timestamp timestamp()	Returns the current date and time of the system time zone.
date date()	Returns the current UTC date based on the current system.
time time()	Returns the current UTC time based on the current system.
datetime datetime()	Returns the current UTC date and time based on the current system.

The `date()`, `time()`, and `datetime()` functions accept three kind of parameters, namely empty, string, and map. The `timestamp()` function accepts two kind of parameters, namely empty and string.

OpenCypher compatibility

- Time in openCypher is measured in milliseconds.
- Time in nGQL is measured in seconds. The milliseconds are displayed in `000`.

Examples

```
> RETURN now(), timestamp(), date(), time(), datetime(), timestamp();
+-----+-----+-----+-----+-----+
| now() | timestamp() | date() | time() | datetime() |
+-----+-----+-----+-----+
| 1625470028 | 1625470028 | 2021-07-05 | 07:27:07.944000 | 2021-07-05T07:27:07.944000 |
+-----+-----+-----+-----+
```

Last update: July 19, 2021

4.5.4 Schema functions

Note

The functions introduced in this topic applies to compatible statements in openCypher only.

Nebula Graph supports the following built-in schema functions:

Function	Description
id(vertex)	Returns the id of a vertex. The data type of the result is the same as the vertex ID.
list tags(vertex)	Returns the tags of a vertex.
list labels(vertex)	Returns the tags of a vertex.
map properties(vertex_or_edge)	Takes in a vertex or an edge and returns its properties.
string type(edge)	Returns the edge type of an edge.
vertex startNode(path)	Takes in an edge or a path and returns its source vertex ID.
string endNode(path)	Takes in an edge or a path and returns its destination vertex ID.
int rank(edge)	Returns the rank value of an edge.

Examples

Last update: August 23, 2021

4.5.5 CASE expressions

The `CASE` expression uses conditions to filter the result of an nGQL query statement. It is usually used in the `YIELD` and `RETURN` clauses. nGQL provides two forms of `CASE` expressions just like openCypher: the simple form and the generic form.

The `CASE` expression will traverse all the conditions. When the first condition is met, the `CASE` expression stops reading the conditions and returns the result. If no conditions are met, it returns the result in the `ELSE` clause. If there is no `ELSE` clause and no conditions are met, it returns `NULL`.

The simple form of CASE expressions

SYNTAX

```
CASE <comparer>
WHEN <value> THEN <result>
[WHEN ...]
[ELSE <default>]
END
```

⚠ Caution

Always remember to end the `CASE` expression with an `END`.

Parameter	Description
<code>comparer</code>	A value or a valid expression that outputs a value. This value is used to compare with the <code>value</code> .
<code>value</code>	It will be compared with the <code>comparer</code> . If the <code>value</code> matches the <code>comparer</code> , then this condition is met.
<code>result</code>	The <code>result</code> is returned by the <code>CASE</code> expression if the <code>value</code> matches the <code>comparer</code> .
<code>default</code>	The <code>default</code> is returned by the <code>CASE</code> expression if no conditions are met.

EXAMPLES

```
nebula> RETURN \
    CASE 2+3 \
    WHEN 4 THEN 0 \
    WHEN 5 THEN 1 \
    ELSE -1 \
    END \
    AS result;
+-----+
| result |
+-----+
| 1      |
+-----+
```

```
nebula> GO FROM "player100" OVER follow \
    YIELD $$.player.name AS Name, \
    CASE $$.player.age > 35 \
    WHEN true THEN "Yes" \
    WHEN false THEN "No" \
    ELSE "Nah" \
    END \
    AS Age_above_35;
+-----+-----+
| Name      | Age_above_35 |
+-----+-----+
| "Tony Parker" | "Yes"      |
+-----+-----+
| "LaMarcus Aldridge" | "No"      |
+-----+-----+
```

The generic form of CASE expressions

SYNTAX

```
CASE
WHEN <condition> THEN <result>
```

```
[WHEN ...]
[ELSE <default>]
END
```

Parameter	Description
condition	If the condition is evaluated as true, the result is returned by the CASE expression.
result	The result is returned by the CASE expression if the condition is evaluated as true.
default	The default is returned by the CASE expression if no conditions are met.

EXAMPLES

```
nebula> YIELD \
  CASE WHEN 4 > 5 THEN 0 \
  WHEN 3+4==7 THEN 1 \
  ELSE 2 \
  END \
  AS result;
+-----+
| result |
+-----+
| 1      |
+-----+
```

```
nebula> MATCH (v:player) WHERE v.age > 30 \
  RETURN v.name AS Name, \
  CASE \
  WHEN v.name STARTS WITH "T" THEN "Yes" \
  ELSE "No" \
  END \
  AS Starts_with_T;
+-----+-----+
| Name      | Starts_with_T |
+-----+-----+
| "Tim"      | "Yes"      |
+-----+-----+
| "LaMarcus Aldridge" | "No"      |
+-----+-----+
| "Tony Parker" | "Yes"      |
+-----+-----+
```

Differences between the simple form and the generic form

To avoid the misuse of the simple form and the generic form, it is important to understand their differences. The following example can help explain them.

```
nebula> GO FROM "player100" OVER follow \
  YIELD $$.player.name AS Name, $$.player.age AS Age, \
  CASE $$.player.age \
  WHEN $$.player.age > 35 THEN "Yes" \
  ELSE "No" \
  END \
  AS Age_above_35;
+-----+-----+
| Name      | Age | Age_above_35 |
+-----+-----+
| "Tony Parker" | 36 | "No"      |
+-----+-----+
| "LaMarcus Aldridge" | 33 | "No"      |
+-----+-----+
```

The preceding GO query is intended to output Yes when the player's age is above 35. However, in this example, when the player's age is 36, the actual output is not as expected: It is No instead of Yes.

This is because the query uses the CASE expression in the simple form, and a comparison between the values of \$\$.player.age and \$\$.player.age > 35 is made. When the player age is 36:

- The value of \$\$.player.age is 36. It is an integer.
- \$\$.player.age > 35 is evaluated to be true. It is a boolean.

The values of \$\$.player.age and \$\$.player.age > 35 do not match. Therefore, the condition is not met and No is returned.

Last update: July 19, 2021

4.5.6 List functions

Nebula Graph supports the following list functions:

Function	Description
keys(expr)	Returns a list containing the string representations for all the property names of vertices, edges, or maps.
labels(vertex)	Returns the list containing all the tags of a vertex.
nodes(path)	Returns the list containing all the vertices in a path.
range(start, end [, step])	Returns the list containing all the fixed-length steps in <code>[start, end]</code> . <code>step</code> is 1 by default.
relationships(path)	Returns the list containing all the relationships in a path.
reverse(list)	Returns the list reversing the order of all elements in the original list.
tail(list)	Returns all the elements of the original list, excluding the first one.
head(list)	Returns the first element of a list.
last(list)	Returns the last element of a list.
coalesce(list)	Returns the first not null value in a list.
reduce()	See reduce() function .

Note

If the argument is `NULL`, the output is undefined.

Examples

```
| [[:follow "player100"->"player125" @0 {degree: 95}], [:serve "player125"->"team204" @0 {end_year: 2018, start_year: 2002}]] |  
+-----+
```

Last update: July 19, 2021

4.5.7 count() function

The `count()` function counts the number of the specified values or rows.

- (Native nGQL) You can use `count()` and `GROUP BY` together to group and count the number of specific values. Use `YIELD` to return.
- (OpenCypher style) You can use `count()` and `RETURN`. `GROUP BY` is not necessary.

Syntax

```
count({expr | *})
```

- `count(*)` returns the number of rows (including NULL).
- `count(expr)` return the number of non-NULL values that meet the expression.
- `count()` and `size()` are different.

EXAMPLES

```
nebula> WITH [NULL, 1, 1, 2, 2] AS a UNWIND a AS b \
    RETURN count(b), count(*), count(DISTINCT b);
```

count(b)	count(*)	count(distinct b)
4	5	2

```
# The statement in the following example searches for the people whom `player101` follows and people who follow `player101`, i.e. a bidirectional query.
nebula> GO FROM "player101" OVER follow BIDIRECT \
    YIELD $$.player.name AS Name \
    | GROUP BY $$.Name YIELD $$.Name, count(*);
```

\$.Name	count(*)
"Dejounte Murray"	1
"LaMarcus Aldridge"	2
"Tim Duncan"	2
"Marco Belinelli"	1
"Manu Ginobili"	1
"Boris Diaw"	1

The preceding example retrieves two columns:

- `$.Name` : the names of the people.
- `count(*)` : how many times the names show up.

Because there are no duplicate names in the `basketballplayer` dataset, the number `2` in the column `count(*)` shows that the person in that row and `player101` have followed each other.

```
# a: The statement in the following example retrieves the age distribution of the players in the dataset.
nebula> LOOKUP ON player \
    YIELD player.age AS playerage \
    | GROUP BY $$.playerage \
    YIELD $$.playerage AS age, count(*) AS number \
    | ORDER BY number DESC, age DESC;
```

age	number
34	4
33	4
30	4
29	4
38	3

```

+-----+-----+
...
# b: The statement in the following example retrieves the age distribution of the players in the dataset.
nebula> MATCH (n:player) \
    RETURN n.age as age, count(*) as number \
    ORDER BY number DESC, age DESC;
+-----+-----+
| age | number |
+-----+-----+
| 34  | 4      |
+-----+-----+
| 33  | 4      |
+-----+-----+
| 30  | 4      |
+-----+-----+
| 29  | 4      |
+-----+-----+
| 38  | 3      |
+-----+-----+
...
+-----+-----+
| count(distinct v2) |
+-----+-----+
| 11          |
+-----+-----+
# The statement in the following example counts the number of edges that Tim Duncan relates.
nebula> MATCH (v:player{name:"Tim Duncan"}) -- (v2) \
    RETURN count(DISTINCT v2);
+-----+-----+
| count(distinct v2) |
+-----+-----+
| 11          |
+-----+-----+
# The statement in the following example counts the number of edges that Tim Duncan relates and returns two columns (no DISTINCT and DISTINCT) in multi-hop queries.
nebula> MATCH (n:player {name : "Tim Duncan"})-[]->(friend:player)-[]->(fof:player) \
    RETURN count(fof), count(DISTINCT fof);
+-----+-----+
| count(fof) | count(distinct fof) |
+-----+-----+
| 4          | 3          |
+-----+-----+

```

.....

Last update: July 19, 2021

4.5.8 collect()

The `collect()` function returns a list containing the values returned by an expression. Using this function aggregates data by merging multiple records or values into a single list.

The aggregate function `collect()` works like `GROUP BY` in SQL.

Examples

```
nebula> UNWIND [1, 2, 1] AS a \
    RETURN a;
+---+
| a |
+---+
| 1 |
+---+
| 2 |
+---+
| 1 |
+---+  
  
nebula> UNWIND [1, 2, 1] AS a \
    RETURN collect(a);
+-----+
| collect(a) |
+-----+
| [1, 2, 1] |
+-----+  
  
nebula> UNWIND [1, 2, 1] AS a \
    RETURN a, collect(a), size(collect(a));
+-----+-----+
| a | collect(a) | size(COLLECT(a)) |
+-----+-----+
| 2 | [2]      | 1           |
+-----+-----+
| 1 | [1, 1]   | 2           |
+-----+-----+  
  
# The following examples sort the results in descending order, limit output rows to 3, and collect the output into a list.©
nebula> UNWIND ["c", "b", "a", "d"] AS p \
    WITH p AS q \
    ORDER BY q DESC LIMIT 3 \
    RETURN collect(q);
+-----+
| collect(q) |
+-----+
| ["d", "c", "b"] |
+-----+  
  
nebula> WITH [1, 1, 2, 2] AS coll \
    UNWIND coll AS x \
    WITH DISTINCT x \
    RETURN collect(x) AS ss;
+-----+
| ss   |
+-----+
| [1, 2] |
+-----+  
  
nebula> MATCH (n:player) \
    RETURN collect(n.age);
+-----+
| collect(n.age) |
+-----+
| [32, 32, 34, 29, 41, 40, 33, 25, 40, 37, ... |
+-----+  
  
# The following example aggregates all the players' names by their ages.
nebula> MATCH (n:player) \
    RETURN n.age AS age, collect(n.name);
+-----+
| age | collect(n.name) |
+-----+
| 24  | ["Giannis Antetokounmpo"] |
+-----+
| 20  | ["Luka Doncic"] |
+-----+
| 25  | ["Joel Embiid", "Kyle Anderson"] |
+-----+
| ... |  
...
```

Last update: July 19, 2021

4.5.9 reduce() function

This topic will describe the `reduce` function.

OpenCypher Compatibility

In openCypher, the `reduce()` function is not defined. nGQL will implement the `reduce()` function in the Cypher way.

Syntax

The `reduce()` function applies an expression to each element in a list one by one, chains the result to the next iteration by taking it as the initial value, and returns the final result. This function iterates each element `e` in the given list, runs the expression on `e`, accumulates the result with the initial value, and store the new result in the accumulator as the initial value of the next iteration. It works like the `fold` or `reduce` method in functional languages such as Lisp and Scala.

```
reduce(<accumulator> = <initial>, <variable> IN <list> | <expression>)
```

Parameter	Description
accumulator	A variable that will hold the accumulated results as the list is iterated.
initial	An expression that runs once to give an initial value to the <code>accumulator</code> .
variable	A variable in the list that will be applied to the expression successively.
list	A list or a list of expressions.
expression	This expression will be run on each element in the list once and store the result value in the <code>accumulator</code> .

Note

The type of the value returned depends on the parameters provided, along with the semantics of the expression.

Examples

```
nebula> RETURN reduce(totalNum = 10, n IN range(1, 3) | totalNum + n) AS r;
+---+
| r |
+---+
| 16 |
+---+  
  
nebula> RETURN reduce(totalNum = -4 * 5, n IN [1, 2] | totalNum + n * 2) AS r;
+---+
| r |
+---+
| -14 |
+---+  
  
nebula> MATCH p = (n:player{name:"LeBron James"})->[:-follow]-(m) \
    RETURN nodes(p)[0].age AS src1, nodes(p)[1].age AS dst2, \
    reduce(totalAge = 100, n IN nodes(p) | totalAge + n.age) AS sum;
+-----+-----+-----+
| src1 | dst2 | sum |
+-----+-----+-----+
| 34   | 31   | 165 |
+-----+-----+-----+
| 34   | 29   | 163 |
+-----+-----+-----+
| 34   | 33   | 167 |
+-----+-----+-----+
| 34   | 26   | 160 |
+-----+-----+-----+
| 34   | 34   | 168 |
+-----+-----+-----+
| 34   | 37   | 171 |
+-----+-----+-----+  
  
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
    | GO FROM $-VertexID over follow \
    WHERE follow.degree != reduce(totalNum = 5, n IN range(1, 3) | $$.player.age + totalNum + n) \
```

```
YIELD $$player.name AS id, $$player.age AS age, follow.degree AS degree;
+-----+-----+-----+
| id      | age | degree |
+-----+-----+-----+
| "Tim Duncan" | 42 | 95 |
+-----+-----+-----+
| "LaMarcus Aldridge" | 33 | 90 |
+-----+-----+-----+
| "Manu Ginobili" | 41 | 95 |
+-----+-----+-----+
```

.....

Last update: July 19, 2021

4.5.10 hash function

The `hash()` function returns the hash value of the argument. The argument can be a number, a string, a list, a boolean, null, or an expression that evaluates to a value of the preceding data types.

The source code of the `hash()` function (MurmurHash2), seed (`0xc70f6907UL`), and other parameters can be found in [MurmurHash2.h](#) .

For Java, the hash function operates as follows.

```
MurmurHash2.hash64("to_be_hashed".getBytes(),"to_be_hashed".getBytes().length, 0xc70f6907)
```

Legacy version compatibility

In nGQL 1.0, when nGQL does not support string VIDs, a common practice is to hash the strings first and then use the values as VIDs. But in nGQL 2.0, both string VIDs and integer VIDs are supported, so there is no need to use `hash()` to set VIDs.

Hash a number

```
nebula> YIELD hash(-123);
+-----+
| hash(-(123)) |
+-----+
| -123         |
+-----+
```

Hash a string

```
nebula> YIELD hash("to_be_hashed");
+-----+
| hash(to_be_hashed)  |
+-----+
| -1098333533029391540 |
+-----+
```

Hash a list

```
nebula> YIELD hash([1,2,3]);
+-----+
| hash([1,2,3])  |
+-----+
| 11093822460243 |
+-----+
```

Hash a boolean

```
nebula> YIELD hash(true);
+-----+
| hash(true)  |
+-----+
| 1           |
+-----+
nebula> YIELD hash(false);
+-----+
| hash(false)  |
+-----+
| 0           |
+-----+
```

Hash NULL

```
nebula> YIELD hash(NULL);
+-----+
| hash(NULL) |
+-----+
| -1         |
+-----+
```

Hash an expression

```
nebula> YIELD hash(toLower("HELLO NEBULA"));
+-----+
| hash(toLower("HELLO NEBULA")) |
+-----+
| -8481157362655072082        |
+-----+
```

.....

Last update: July 19, 2021

4.5.11 Predicate functions

Predicate functions return `true` or `false`. They are most commonly used in `WHERE` clauses.

Nebula Graph supports the following predicate functions:

Functions	Description
<code>exists()</code>	Returns <code>true</code> if the specified property exists in the vertex, edge or map. Otherwise, returns <code>false</code> .
<code>any()</code>	Returns <code>true</code> if the specified predicate holds for at least one element in the given list. Otherwise, returns <code>false</code> .
<code>all()</code>	Returns <code>true</code> if the specified predicate holds for all elements in the given list. Otherwise, returns <code>false</code> .
<code>none()</code>	Returns <code>true</code> if the specified predicate holds for no element in the given list. Otherwise, returns <code>false</code> .
<code>single()</code>	Returns <code>true</code> if the specified predicate holds for exactly one of the elements in the given list. Otherwise, returns <code>false</code> .

Note

NULL is returned if the list is NULL or all of its elements are NULL.

Compatibility

In openCypher, only function `exists()` is defined and specified. The other functions are implement-dependent.

Syntax

```
<predicate>(<variable> IN <list> WHERE <condition>)
```

Examples

```
nebula> RETURN any(n IN [1, 2, 3, 4, 5, NULL] \
    WHERE n > 2) AS r;
+-----+
| r   |
+-----+
| true |
+-----+

nebula> RETURN single(n IN range(1, 5) \
    WHERE n == 3) AS r;
+-----+
| r   |
+-----+
| true |
+-----+

nebula> RETURN none(n IN range(1, 3) \
    WHERE n == 0) AS r;
+-----+
| r   |
+-----+
| true |
+-----+

nebula> WITH [1, 2, 3, 4, 5, NULL] AS a \
    RETURN any(n IN a WHERE n > 2);
+-----+
| any(n IN a WHERE (n>2)) |
+-----+
| true                   |
+-----+

nebula> MATCH p = (n:player{name:"LeBron James"})->[:follow]-(m) \
    RETURN nodes(p)[0].name AS n1, nodes(p)[1].name AS n2, \
    all(n IN nodes(p) WHERE n.name NOT STARTS WITH "D") AS b;
+-----+-----+-----+
```

```

| n1      | n2      | b      |
+-----+-----+-----+
| "LeBron James" | "Danny Green" | false |
+-----+-----+-----+
| "LeBron James" | "Dejounte Murray" | false |
+-----+-----+-----+
| "LeBron James" | "Chris Paul" | true |
+-----+-----+-----+
| "LeBron James" | "Kyrie Irving" | true |
+-----+-----+-----+
| "LeBron James" | "Carmelo Anthony" | true |
+-----+-----+-----+
| "LeBron James" | "Dwyane Wade" | false |
+-----+-----+-----+
nebula> MATCH p = (n:player{name:"LeBron James"})-[:follow]->(m) \
    RETURN single(n IN nodes(p) WHERE n.age > 40) AS b;
+-----+
| b      |
+-----+
| true  |
+-----+
nebula> MATCH (n:player) \
    RETURN exists(n.id), n IS NOT NULL;
+-----+-----+
| exists(n.id) | n IS NOT NULL |
+-----+-----+
| false      | true   |
+-----+-----+
...
nebula> MATCH (n:player) \
    WHERE exists(n['name']) RETURN n;
+-----+
| n      |
+-----+
| ("Grant Hill" :player{age: 46, name: "Grant Hill"}) |
+-----+
| ("Marc Gasol" :player{age: 34, name: "Marc Gasol"}) |
+-----+
...

```

.....

Last update: July 19, 2021

4.5.12 User-defined functions

OpenCypher compatibility

User-defined functions (UDF) and storage processes are not yet supported nor designed in Nebula Graph 2.5.0.

Last update: July 19, 2021

4.6 General queries statements

4.6.1 MATCH

The `MATCH` statement supports searching based on pattern matching.

A `MATCH` statement defines a [search pattern](#) and uses it to match data stored in Nebula Graph and to retrieve them in the form defined in the `RETURN` clause.

The examples in this topic use the [basketballplayer](#) dataset as the sample dataset.

Syntax

The syntax of `MATCH` is relatively more flexible compared with that of other query statements such as `GO` or `LOOKUP`. But generally, it can be summarized as follows.

```
MATCH <pattern> [<WHERE clause>] RETURN <output>
```

The workflow of MATCH

1. The `MATCH` statement uses a native index to locate a source vertex or an edge. The source vertex or the edge can be in any position in the pattern. In other words, in a valid `MATCH` statement, **there must be an indexed property, a tag, or an edge type. Or the VID of a specific vertex must be specified with the `id()` function in the WHERE clause**. For how to create an index, see [create native index](#).
2. The `MATCH` statement searches through the pattern to match edges or vertices.

Note

The path type of the `MATCH` statement is `trail`. That is, only vertices can be repeatedly visited in the graph traversal. Edges cannot be repeatedly visited. For details, see [path](#).

3. The `MATCH` statement retrieves data according to the `RETURN` clause.

OpenCypher compatibility

- For now, nGQL does not support traversing all vertices and edges with `MATCH`, such as `MATCH (v) RETURN v`. However, after the index of a certain tag is created, all corresponding vertices can be traversed, such as `MATCH (v:T1) RETURN v`.
- Graph pattern is not supported in the `WHERE` clause.

Using patterns in MATCH statements

PREREQUISITES

Make sure there is at least one index in the `MATCH` statement, or there is a specified VID. If you want to create an index, but there are already related vertices, edges, or properties, you must rebuild indexes after creating the index to make it valid.

Caution

Correct use of indexes can speed up queries, but indexes can dramatically reduce the write performance. The performance reduction can be as much as 90% or even more. **DO NOT** use indexes in production environments unless you are fully aware of their influences on your service.

```
# The following example creates an index on both the name property of the tag player and the edge type follow.
nebula> CREATE TAG INDEX name ON player(name(20));
```

```

nebula> CREATE EDGE INDEX follow_index on follow();

# The following example rebuilds the index.
nebula> REBUILD TAG INDEX name;
+-----+
| New Job Id |
+-----+
| 121 |
+-----+

nebula> REBUILD EDGE INDEX follow_index
+-----+
| New Job Id |
+-----+
| 122 |
+-----+

# The following example makes sure the index is rebuilt successfully.
nebula> SHOW JOB 121;
+-----+-----+-----+-----+
| Job Id(TaskId) | Command(Dest) | Status | Start Time | Stop Time |
+-----+-----+-----+-----+
| 121 | "REBUILD_TAG_INDEX" | "FINISHED" | 2021-05-27T02:18:02.000 | 2021-05-27T02:18:02.000 |
+-----+-----+-----+-----+
| 0 | "storaged1" | "FINISHED" | 2021-05-27T02:18:02.000 | 2021-05-27T02:18:02.000 |
+-----+-----+-----+-----+
| 1 | "storaged0" | "FINISHED" | 2021-05-27T02:18:02.000 | 2021-05-27T02:18:02.000 |
+-----+-----+-----+-----+
| 2 | "storaged2" | "FINISHED" | 2021-05-27T02:18:02.000 | 2021-05-27T02:18:02.000 |
+-----+-----+-----+-----+

nebula> SHOW JOB 122;
+-----+-----+-----+-----+
| Job Id(TaskId) | Command(Dest) | Status | Start Time | Stop Time |
+-----+-----+-----+-----+
| 122 | "REBUILD_EDGE_INDEX" | "FINISHED" | 2021-05-27T02:18:11.000 | 2021-05-27T02:18:11.000 |
+-----+-----+-----+-----+
| 0 | "storaged1" | "FINISHED" | 2021-05-27T02:18:11.000 | 2021-05-27T02:18:21.000 |
+-----+-----+-----+-----+
| 1 | "storaged0" | "FINISHED" | 2021-05-27T02:18:11.000 | 2021-05-27T02:18:21.000 |
+-----+-----+-----+-----+
| 2 | "storaged2" | "FINISHED" | 2021-05-27T02:18:11.000 | 2021-05-27T02:18:21.000 |
+-----+-----+-----+-----+

```

MATCH VERTICES

You can use a user-defined variable in a pair of parentheses to represent a vertex in a pattern. For example: `(v)`.

MATCH TAGS

Note

The prerequisite for matching a tag is that the tag itself has an index or a certain property of the tag has an index. Otherwise, you cannot execute the `MATCH` statement based on the tag.

You can specify a tag with `:<tag_name>` after the vertex in a pattern.

```

nebula> MATCH (v:player) RETURN v;
+-----+
| v |
+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----+
| ("player106" :player{age: 25, name: "Kyle Anderson"}) |
+-----+
| ("player115" :player{age: 40, name: "Kobe Bryant"}) |
+-----+
...

```

MATCH VERTEX PROPERTIES

Note

The prerequisite for matching a vertex property is that the tag itself has an index of the corresponding property. Otherwise, you cannot execute the `MATCH` statement to match the property.

You can specify a vertex property with `{<prop_name>: <prop_value>}` after the tag in a pattern.

```
# The following example uses the name property to match a vertex.
nebula> MATCH (v:player{name:"Tim Duncan"}) RETURN v;
+-----+
| v
+-----+
| ("player100" :player{name: "Tim Duncan", age: 42}) |
+-----+
```

The `WHERE` clause can do the same thing:

```
nebula> MATCH (v:player) WHERE v.name == "Tim Duncan" RETURN v;
+-----+
| v
+-----+
| ("player100" :player{name: "Tim Duncan", age: 42}) |
+-----+
```

OpenCypher compatibility

In OpenCypher 9, `=` is the equality operator. However, in nGQL, `==` is the equality operator and `=` is the assignment operator (as in C++ or Java).

MATCH VIDS

You can use the VID to match a vertex. The `id()` function can retrieve the VID of a vertex.

```
nebula> MATCH (v) WHERE id(v) == 'player101' RETURN v;
+-----+
| v
+-----+
| (player101) player.name:Tony Parker,player.age:36 |
+-----+
```

To match multiple VIDs, use `WHERE id(v) IN [vid_list]`.

```
nebula> MATCH (v:player { name: 'Tim Duncan' })--(v2) \
    WHERE id(v2) IN ["player101", "player102"] RETURN v2;
+-----+
| v2
+-----+
| ("player101" :player{name: "Tony Parker", age: 36}) |
+-----+
| ("player102" :player{name: "LaMarcus Aldridge", age: 33}) |
+-----+
| ("player101" :player{name: "Tony Parker", age: 36}) |
+-----+
```

MATCH CONNECTED VERTICES

You can use the `--` symbol to represent edges of both directions and match vertices connected by these edges.

Legacy

In nGQL 1.x, the `--` symbol is used for inline comments. Starting from nGQL 2.x, the `--` symbol represents an incoming or outgoing edge.

```
nebula> MATCH (v:player{name:"Tim Duncan"})--(v2) \
    RETURN v2.name AS Name;
+-----+
| Name
+-----+
| "Tony Parker"
+-----+
| "LaMarcus Aldridge"
+-----+
| "Marco Belinelli"
+-----+
| "Danny Green"
+-----+
| "Aron Baynes"
+-----+
| ...
+-----+
```

You can add a `>` or `<` to the `--` symbol to specify the direction of an edge.

In the following example, `-->` represents an edge that starts from `v` and points to `v2`. To `v`, this is an outgoing edge, and to `v2` this is an incoming edge.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-->(v2) \
    RETURN v2.name AS Name;
+-----+
| Name |
+-----+
| "Spurs" |
| "Tony Parker" |
| "Manu Ginobili" |
+-----+
```

To extend the pattern, you can add more vertices and edges.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-->(v2)<--(v3) \
    RETURN v3.name AS Name;
+-----+
| Name |
+-----+
| "Tony Parker" |
| "Tiago Splitter" |
| "Dejounte Murray" |
| "Tony Parker" |
| "LaMarcus Aldridge" |
+-----+
...
```

If you do not need to refer to a vertex, you can omit the variable representing it in the parentheses.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-->()<--(v3) \
    RETURN v3.name AS Name;
+-----+
| Name |
+-----+
| "Tony Parker" |
| "LaMarcus Aldridge" |
| "Rudy Gay" |
| "Danny Green" |
| "Kyle Anderson" |
+-----+
...
```

MATCH PATHS

Connected vertices and edges form a path. You can use a user-defined variable to name a path as follows.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-->(v2) \
    RETURN p;
+-----+
| p |
+-----+
| <"player100" :player{age: 42, name: "Tim Duncan"}-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})> |
+-----+
| <"player100" :player{age: 42, name: "Tim Duncan"}-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})> |
+-----+
| <"player100" :player{age: 42, name: "Tim Duncan"}-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})> |
+-----+
```

OpenCypher compatibility

In nGQL, the `@` symbol represents the rank of an edge, but openCypher has no such concept.

MATCH EDGES

Besides using `--`, `-->`, or `<--` to indicate a nameless edge, you can use a user-defined variable in a pair of square brackets to represent a named edge. For example: `-[e]-`.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]-(v2) \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player101"->"player100" @0 {degree: 95}]
+-----+
| [:follow "player102"->"player100" @0 {degree: 75}]
+-----+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}]
+-----+
...
```

MATCH EDGE TYPES

Just like vertices, you can specify edge types with `:<edge_type>` in a pattern. For example: `-[e:follow]-`.

```
nebula> MATCH ()-[e:follow]-() \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player113"->"player119" @0 {degree: 99}]
+-----+
| [:follow "player130"->"player149" @0 {degree: 80}]
+-----+
| [:follow "player149"->"player130" @0 {degree: 80}]
+-----+
| [:follow "player136"->"player117" @0 {degree: 90}]
+-----+
| [:follow "player142"->"player117" @0 {degree: 90}]
+-----+
...
```

MATCH EDGE TYPE PROPERTIES

 **Note**

The prerequisite for matching an edge type property is that the edge type itself has an index of the corresponding property. Otherwise, you cannot execute the `MATCH` statement to match the property.

You can specify edge type properties with `{<prop_name>: <prop_value>}` in a pattern. For example: `[e:follow{likeness:95}]`.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e:follow{degree:95}]->(v2) \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player100"->"player101" @0 {degree: 95}]
+-----+
| [:follow "player100"->"player125" @0 {degree: 95}]
+-----+
```

MATCH MULTIPLE EDGE TYPES

The `|` symbol can help matching multiple edge types. For example: `[e:follow|:serve]`. The English colon (`:`) before the first edge type cannot be omitted, but the English colon before the subsequent edge type can be omitted, such as `[e:follow|serve]`.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e:follow|:serve]->(v2) \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player100"->"player101" @0 {degree: 95}]
+-----+
| [:follow "player100"->"player125" @0 {degree: 95}]
+-----+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}]
+-----+
```

MATCH MULTIPLE EDGES

You can extend a pattern to match multiple edges in a path.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[]->(v2)-[e:serve]-(v3) \
    RETURN v2, v3;
+-----+
| v2
+-----+
| v3
+-----+
```

```

| ("player204" :team{name: "Spurs"}) | ("player101" :player{name: "Tony Parker", age: 36}) |
+-----+-----+
| ("player204" :team{name: "Spurs"}) | ("player102" :player{name: "LaMarcus Aldridge", age: 33}) |
+-----+-----+
| ("player204" :team{name: "Spurs"}) | ("player103" :player{age: 32, name: "Rudy Gay"}) |
+-----+-----+
...

```

MATCH FIXED-LENGTH PATHS

You can use the `:<edge_type>*<hop>` pattern to match a fixed-length path. `hop` must be a non-negative integer.

```

nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \
    RETURN DISTINCT v2 AS Friends;
+-----+
| Friends
+-----+
| ("player100" :player{name: "Tim Duncan", age: 42}) |
+-----+
| ("player102" :player{name: "LaMarcus Aldridge", age: 33}) |
+-----+
| ("player125" :player{name: "Manu Ginobili", age: 41}) |
+-----+

```

If `hop` is 0, the pattern will match the source vertex of the path.

```

nebula> MATCH (v:player{name:"Tim Duncan"}) -[*0]-> (v2) \
    RETURN v2;
+-----+
| v2
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+

```

MATCH VARIABLE-LENGTH PATHS

You can use the `:<edge_type>*[minHop]..<maxHop>` pattern to match variable-length paths.

Parameter	Description
<code>minHop</code>	Optional. It represents the minimum length of the path. <code>minHop</code> must be a non-negative integer. The default value is 1.
<code>maxHop</code>	Required. It represents the maximum length of the path. <code>maxHop</code> must be a non-negative integer. It has no default value.

OpenCypher compatibility

In openCypher, `maxHop` is optional and defaults to infinity. When no bounds are given, `...` can be omitted. However, in nGQL, `maxHop` is required. And `...` cannot be omitted.

```

nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*1..3]->(v2) \
    RETURN v2 AS Friends;
+-----+
| Friends
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+
| ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----+
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |
+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----+

```

You can use the `DISTINCT` keyword to aggregate duplicate results.

```

nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*1..3]->(v2:player) \
    RETURN DISTINCT v2 AS Friends, count(v2);
+-----+-----+
| Friends | count(v2) |
+-----+-----+
| ("player125" :player{age: 41, name: "Manu Ginobili"}) | 3 |
+-----+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) | 1 |
+-----+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | 4 |
+-----+-----+

```

```
+-----+-----+
| ("player101" :player{age: 36, name: "Tony Parker"}) | 3 |
+-----+-----+
```

If `minHop` is `0`, the pattern will match the source vertex of the path. Compared to the preceding statement, the following example uses `0` as the `minHop`. So in the following result set, "Tim Duncan" is counted one more time than it is in the preceding result set because it is the source vertex.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[:follow*0..3]->(v2:player) \
    RETURN DISTINCT v2 AS Friends, count(v2);
+-----+-----+
| Friends | count(v2) |
+-----+-----+
| ("player125" :player{age: 41, name: "Manu Ginobili"}) | 3 |
+-----+-----+
| ("player101" :player{age: 36, name: "Tony Parker"}) | 3 |
+-----+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) | 1 |
+-----+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | 5 |
+-----+-----+
```

MATCH VARIABLE-LENGTH PATHS WITH MULTIPLE EDGE TYPES

You can specify multiple edge types in a fixed-length or variable-length pattern. In this case, `hop`, `minHop`, and `maxHop` take effect on all edge types.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[:follow|serve*2]->(v2) \
    RETURN DISTINCT v2;
+-----+
| v2 |
+-----+
| ("player100" :player{name: "Tim Duncan", age: 42}) |
+-----+
| ("player102" :player{name: "LaMarcus Aldridge", age: 33}) |
+-----+
| ("player125" :player{name: "Manu Ginobili", age: 41}) |
+-----+
| ("player204" :team{name: "Spurs"}) |
+-----+
| ("player215" :team{name: "Hornets"}) |
+-----+
```

Common retrieving operations

RETRIEVE VERTEX OR EDGE INFORMATION

Use `RETURN {<vertex_name> | <edge_name>}` to retrieve all the information of a vertex or an edge.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN v;
+-----+
| v |
+-----+
| ("player100" :player{name: "Tim Duncan", age: 42}) |
+-----+
nebula> MATCH (v:player{name:"Tim Duncan"})-[:e]->(v2) \
    RETURN e;
+-----+
| e |
+-----+
| [:follow "player100"-->"player101" @0 {degree: 95}] |
+-----+
| [:follow "player100"-->"player125" @0 {degree: 95}] |
+-----+
| [:serve "player100"-->"team204" @0 {end_year: 2016, start_year: 1997}] |
+-----+
```

RETRIEVE VIDS

Use the `id()` function to retrieve VIDs.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN id(v);
+-----+
| id(v) |
+-----+
| "player100" |
+-----+
```

RETRIEVE TAGS

Use the `labels()` function to retrieve the list of tags on a vertex.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN labels(v);
+-----+
| labels(v) |
+-----+
| ["player"] |
+-----+
```

To retrieve the *n*th element in the `labels(v)` list, use `labels(v)[n-1]`. The following example shows how to use `labels(v)[0]` to retrieve the first tag in the list.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN labels(v)[0];
+-----+
| labels(v)[0] |
+-----+
| "player" |
+-----+
```

RETRIEVE A SINGLE PROPERTY ON A VERTEX OR AN EDGE

Use `RETURN {<vertex_name> | <edge_name>}.<property>` to retrieve a single property.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN v.age;
+-----+
| v.age |
+-----+
| 42    |
+-----+
```

Use `AS` to specify an alias for a property.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN v.age AS Age;
+-----+
| Age |
+-----+
| 42  |
+-----+
```

RETRIEVE ALL PROPERTIES ON A VERTEX OR AN EDGE

Use the `properties()` function to retrieve all properties on a vertex or an edge.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
    RETURN properties(v2);
+-----+
| properties(v2) |
+-----+
| {"name":"Spurs"} |
+-----+
| {"name":"Tony Parker", "age":36} |
+-----+
| {"age":41, "name":"Manu Ginobili"} |
+-----+
```

RETRIEVE EDGE TYPES

Use the `type()` function to retrieve the matched edge types.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e]->() \
    RETURN DISTINCT type(e);
+-----+
| type(e) |
+-----+
| "follow" |
+-----+
| "serve" |
+-----+
```

RETRIEVE PATHS

Use `RETURN <path_name>` to retrieve all the information of the matched paths.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[*3]->() \
    RETURN p;
+-----+
| p
+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})-[:serve@0 {end_year: 2019, start_year: 2015}]->("team204" :team{name: "Spurs"})> |
+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})-[:serve@0 {end_year: 2015, start_year: 2006}]->("team203" :team{name: "Trail Blazers"})> |
+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})-[:follow@0 {degree: 75}]->("player101" :player{age: 36, name: "Tony Parker"})> |
+-----+
...
```

RETRIEVE VERTICES IN A PATH

Use the `nodes()` function to retrieve all vertices in a path.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
    RETURN nodes(p);
+-----+
| nodes(p)
+-----+
| [{"player100" :star{} :player{age: 42, name: "Tim Duncan"}}, {"player204" :team{name: "Spurs"}}, {"player101" :player{name: "Tony Parker", age: 36}}, {"player125" :player{name: "Manu Ginobili", age: 41}}]
+-----+
```

RETRIEVE EDGES IN A PATH

Use the `relationships()` function to retrieve all edges in a path.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
    RETURN relationships(p);
+-----+
| relationships(p)
+-----+
| [{"follow": "player100" -> "player101" @0 {degree: 95}}, {"follow": "player100" -> "player125" @0 {degree: 95}}, {"serve": "player100" -> "team204" @0 {end_year: 2016, start_year: 1997}}]
+-----+
```

RETRIEVE PATH LENGTH

Use the `length()` function to retrieve the length of a path.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[*..2]->(v2) \
    RETURN p AS Paths, length(p) AS Length;
+-----+-----+
| Paths | Length |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})-[:serve@0 {end_year: 2018, start_year: 2002}]->("team204" :team{name: "Spurs"})> | 2 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})-[:follow@0 {degree: 90}]->("player100" :player{age: 42, name: "Tim Duncan"})> | 2 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:serve@0 {end_year: 2019, start_year: 2018}]->("team215" :team{name: "Hornets"})> | 2 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:serve@0 {end_year: 2018, start_year: 1999}]->("team204" :team{name: "Spurs"})> | 2 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})> | 2 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})> | 2 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 95}]->("player100" :player{age: 42, name: "Tim Duncan"})> | 2 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})> | 1 |
+-----+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})> | 1 |
+-----+-----+
```

Performance tips

In Nebula Graph 2.5.0, the `MATCH` statement is not optimized for resource usage and performance. Simpler operations can be replaced by `GO`, `LOOKUP`, `|`, and `FETCH`.

Last update: August 4, 2021

4.6.2 LOOKUP

The `LOOKUP` statement traverses data based on indexes. You can use `LOOKUP` for the following purposes:

- Search for the specific data based on conditions defined by the `WHERE` clause.
- List vertices with a tag: retrieve the VID of all vertices with a tag.
- List edges with an edge type: retrieve the source vertex IDs, destination vertex IDs, and ranks of all edges with an edge type.
- Count the number of vertices or edges with a tag or an edge type.

OpenCypher compatibility

This topic applies to native nGQL only.

Prerequisites

Before using the `LOOKUP` statement, make sure that at least one index is created. If there are already related vertices, edges, or properties before an index is created, the user must [rebuild native index](#) after creating the index to make it valid.

⚠ Caution

Correct use of indexes can speed up queries, but indexes can dramatically reduce the write performance. The performance reduction can be as much as 90% or even more. **DO NOT** use indexes in production environments unless you are fully aware of their influences on your service.

Syntax

```
LOOKUP ON {<vertex_tag> | <edge_type>} [WHERE <expression> [AND <expression> ...]] [YIELD <return_list>];
<return_list>
  <prop_name> [AS <col_alias>] [, <prop_name> [AS <prop_alias>] ...];
```

- `WHERE <expression>`: filters data with specified conditions. Both `AND` and `OR` are supported between different expressions. For more information, see [WHERE](#).
- `YIELD <return_list>`: specifies the results to be returned and the format of the results.
- If there is a `WHERE` clause but no `YIELD` clause:
 - The Vertex ID is returned when you `LOOKUP` a tag.
 - The source vertex ID, destination vertex ID, and rank of the edge are returned when `LOOKUP` an edge type.

Limitations of using WHERE in LOOKUP

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

- `$-` and `$^`.
- Nested AliasProp expressions in operation expressions and function expressions are not supported.
- Range scan is not supported in the string-type index.
- The `XOR` and `NOT` operations are not supported.

Retrieve Vertices

The following example returns vertices whose `name` is `Tony Parker` and the tag is `player`.

```

nebula> CREATE TAG INDEX index_player ON player(name(30), age);

nebula> REBUILD TAG INDEX index_player;
+-----+
| New Job Id |
+-----+
| 15          |
+-----+

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

# The following example uses regex to retrieve vertices.
nebula> LOOKUP ON player WHERE player.name =~ "^.{15,20}$" \
    YIELD player.name, player.age;
+-----+-----+
| VertexID | player.name | player.age |
+-----+-----+
| "player147" | "Amar'e Stoudemire" | 36 |
+-----+-----+
| "player144" | "Shaquille O'Neal" | 47 |
+-----+-----+
...
.

nebula> LOOKUP ON player WHERE player.name CONTAINS toLower("L") \
    YIELD player.name, player.age;
+-----+-----+
| VertexID | player.name | player.age |
+-----+-----+
| "player145" | "JaVale McGee" | 31 |
+-----+-----+
| "player144" | "Shaquille O'Neal" | 47 |
+-----+-----+
| "player102" | "LaMarcus Aldridge" | 33 |
+-----+-----+
...
.

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> REBUILD EDGE INDEX index_follow;
+-----+
| New Job Id |
+-----+
| 62          |
+-----+

nebula> LOOKUP ON follow WHERE follow.degree == 90;
+-----+-----+
| SrcVID | DstVID | Ranking |
+-----+-----+
| "player101" | "player102" | 0 |
+-----+-----+
| "player133" | "player114" | 0 |
+-----+-----+
| "player133" | "player144" | 0 |
+-----+-----+
...
.

nebula> LOOKUP ON follow WHERE follow.degree == 90 YIELD follow.degree;
+-----+-----+-----+
| SrcVID | DstVID | Ranking | follow.degree |
+-----+-----+-----+
| "player101" | "player102" | 0 | 90 |
+-----+-----+-----+
| "player133" | "player114" | 0 | 90 |
+-----+-----+-----+
| "player133" | "player144" | 0 | 90 |
+-----+-----+-----+
...
.

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 |
+-----+-----+-----+
| "player105" | 2010 | 2018 | "Spurs" |
+-----+-----+-----+
| "player105" | 2009 | 2010 | "Cavaliers" |
+-----+-----+-----+
| "player105" | 2018 | 2019 | "Raptors" |
+-----+-----+-----+
```

List vertices or edges with a tag or an edge type

To list vertices or edges with a tag or an edge type, at least one index must exist on the tag, the edge type, or its property.

For example, if there is a `player` tag with a `name` property and an `age` property, to retrieve the VID of all vertices tagged with `player`, there has to be an index on the `player` tag itself, the `name` property, or the `age` property.

- The following example shows how to retrieve the VID of all vertices tagged with `player`.

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

nebula> REBUILD TAG INDEX player_index;
+-----+
| New Job Id |
+-----+
| 66 |
+-----+

nebula> INSERT VERTEX player(name,age) VALUES "player100":("Tim Duncan", 42), "player101":("Tony Parker", 36);

# The following statement retrieves the VID of all vertices with the tag of player. It is similar to MATCH (n:player) RETURN id(n) /*, n */.

nebula> LOOKUP ON player;
+-----+
| _vid |
+-----+
| "player100" |
+-----+
| "player101" |
+-----+
```

- The following example shows how to retrieve the source Vertex IDs, destination vertex IDs, and ranks of all edges of the `like` edge type.

```
nebula> CREATE EDGE like(likeness int);
nebula> CREATE EDGE INDEX like_index on like();

nebula> REBUILD EDGE INDEX like_index;
+-----+
| New Job Id |
+-----+
| 88 |
+-----+

nebula> INSERT EDGE like(likeness) values "player100"->"player101":(95);

# The following statement retrieves all edges with the edge type of like. It is similar to MATCH (s)-[e:like]->(d) RETURN id(s), rank(e), id(d) /*, type(e) */.

nebula> LOOKUP ON like;
+-----+-----+-----+
| _src | _ranking | _dst |
+-----+-----+-----+
| "player100" | 0 | "player101" |
+-----+-----+-----+
```

Count the numbers of vertices or edges

The following example shows how to count the number of vertices tagged with `player` and edges of the `like` edge type.

```
nebula> LOOKUP ON player | YIELD COUNT(*) AS Player_Number;
+-----+
| Player_Number |
+-----+
| 2 |
+-----+

nebula> LOOKUP ON like | YIELD COUNT(*) AS Like_Number;
+-----+
```

Like_Number
1

Note

You can also use `show-stats` to count the numbers of vertices or edges.

Last update: August 4, 2021

4.6.3 GO

GO traverses in a graph with specified filters and returns results.

OpenCypher compatibility

This topic applies to native nGQL only.

Syntax

```
GO [[<M> TO] <N> STEPS ] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
[YIELD [DISTINCT] <return_list>]
[] ORDER BY <expression> [{ASC | DESC}]]
[ | LIMIT [<offset_value>,] <number_rows>]

GO [[<M> TO] <N> STEPS ] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
[ | GROUP BY {col_name | expr | position} YIELD <col_name>]

<vertex_list> ::= 
  <vid> [, <vid> ...]

<edge_type_list> ::= 
  edge_type [, edge_type ...]
  | *
```

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

- `<N> STEPS` : specifies the hop number. If not specified, the default value for `N` is `one`. When `N` is `zero`, Nebula Graph does not traverse any edges and returns nothing.

Note

The path type of the `GO` statement is `walk`, which means both vertices and edges can be repeatedly visited in graph traversal. For more information, see [Path](#).

- `M TO N STEPS` : traverses from `M` to `N` hops. When `M` is `zero`, the output is the same as that of `M` is `one`. That is, the output of `GO 0 TO 2` and `GO 1 TO 2` are the same.
- `<vertex_list>` : represents a list of vertex IDs separated by commas, or a special place holder `$-.id`. For more information, see [Pipe](#).
- `<edge_type_list>` : represents a list of edge types which the traversal can go through.
- `REVERSELY | BIDIRECT` : defines the direction of the query. By default, the `GO` statement searches for outgoing edges of `<vertex_list>`. If `REVERSELY` is set, `GO` searches for incoming edges. If `BIDIRECT` is set, `GO` searches for edges of both directions.
- `WHERE <expression>` : specifies the traversal filters. You can use the `WHERE` clause for the source vertices, the edges, and the destination vertices. You can use it together with `AND`, `OR`, `NOT`, and `XOR`. For more information, see [WHERE](#).

Note

There are some restrictions for the `WHERE` clause when you traverse along with multiple edge types. For example, `WHERE edge1.prop1 > edge2.prop2` is not supported.

- `YIELD [DISTINCT] <return_list>` : specifies the desired output. For more information, see [YIELD](#). When not specified, the destination vertex IDs will be returned by default.
- `ORDER BY` : sorts outputs with specified orders. For more information, see [ORDER BY](#).

Note

When the sorting method is not specified, the output orders can be different for the same query.

- `LIMIT` : limits the number of rows of the output. For more information, see [LIMIT](#).
- `GROUP BY` : groups outputs into subgroups based on values of the specified properties. For more information, see [GROUP BY](#).

Examples

```
# The following example returns the teams that player 102 serves.  
nebula> GO FROM "player102" \  
      OVER serve;  
+-----+  
| serve._dst |  
+-----+  
| "team203" |  
+-----+  
| "team204" |  
+-----+
```

```
# The following example returns the friends of player 102 with 2 hops.  
nebula> GO 2 STEPS FROM "player102" \  
      OVER follow;  
+-----+  
| follow._dst |  
+-----+  
| "player101" |  
+-----+
```

```

| "player125" |
+-----+
...

# The following example adds a filter for the traversal.
nebula> GO FROM "player100", "player102" \
    OVER serve \
    WHERE serve.start_year > 1995 \
    YIELD DISTINCT $$ .team.name AS team_name, serve.start_year AS start_year, $$ .player.name AS player_name;
+-----+-----+-----+
| team_name | start_year | player_name |
+-----+-----+-----+
| "Spurs" | 1997 | "Tim Duncan" |
+-----+-----+-----+
| "Trail Blazers" | 2006 | "LaMarcus Aldridge" |
+-----+-----+-----+
| "Spurs" | 2015 | "LaMarcus Aldridge" |
+-----+-----+-----+


# The following example traverses along with multiple edge types. If there is no value for a property, the output is __EMPTY__.
nebula> GO FROM "player100" \
    OVER follow, serve \
    YIELD follow.degree, serve.start_year;
+-----+-----+
| follow.degree | serve.start_year |
+-----+-----+
| 95 | __EMPTY__ |
+-----+-----+
| 95 | __EMPTY__ |
+-----+-----+
| __EMPTY__ | 1997 |
+-----+-----+


# The following example returns the incoming edges of player 100.
nebula> GO FROM "player100" \
    OVER follow REVERSELY \
    YIELD follow._dst AS destination;
+-----+
| destination |
+-----+
| "player101" |
+-----+
| "player102" |
+-----+
...

# This MATCH query shares the same semantics with the preceding GO query.
nebula> MATCH (v)<-[e:follow]- (v2) \
    WHERE id(v) == 'player100' \
    RETURN id(v2) AS destination;
+-----+
| destination |
+-----+
| "player101" |
+-----+
| "player102" |
+-----+
...

# The following example retrieves the friends of player 100 and the teams that they serve.
nebula> GO FROM "player100" \
    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" | "Spurs" |
+-----+-----+
| "Tony Parker" | "Hornets" |
+-----+-----+
...

# This MATCH query shares the same semantics with the preceding GO query.
nebula> MATCH (v)<-[e:follow]- (v2)-[e2:serve]->(v3) \
    WHERE id(v) == 'player100' \
    RETURN v2.name AS FriendOf, v3.name AS Team;
+-----+-----+
| FriendOf | Team |
+-----+-----+
| "Tony Parker" | "Spurs" |
+-----+-----+
| "Tony Parker" | "Hornets" |
+-----+-----+
...

# The following example returns the outgoing edges and the incoming edges of player 102.
nebula> GO FROM "player102" \

```

```

        OVER follow BIDIRECT \
        YIELD follow._dst AS both;
+-----+
| both      |
+-----+
| "player100" |
+-----+
| "player101" |
+-----+
...
.

# This MATCH query shares the same semantics with the preceding GO query.
nebula> MATCH (v) -[e:follow]->(v2) \
    WHERE id(v)== "player102" \
    RETURN id(v2) AS both;
+-----+
| both      |
+-----+
| "player101" |
+-----+
| "player103" |
+-----+
...
.

# The following example retrieves the friends of player 100 within 1 or 2 hops.
nebula> GO 1 TO 2 STEPS FROM "player100" \
    OVER follow \
    YIELD follow._dst AS destination;
+-----+
| destination |
+-----+
| "player101" |
+-----+
| "player125" |
+-----+
...
.

# This MATCH query shares the same semantics with the preceding GO query.
nebula> MATCH (v) -[e:follow*1..2]->(v2) \
    WHERE id(v) == "player100" \
    RETURN id(v2) AS destination;
+-----+
| destination |
+-----+
| "player100" |
+-----+
| "player102" |
+-----+
...
.

# The following example the outputs according to age.
nebula> GO 2 STEPS FROM "player100" \
    OVER follow \
    YIELD follow._src AS src, follow._dst AS dst, $$._player.age AS age |\
    GROUP BY $._dst \
    YIELD $._dst AS dst, collect_set($._src) AS src, collect($._age) AS age;
+-----+-----+-----+
| dst      | src      | age      |
+-----+-----+-----+
| "player125" | ["player101"] | [41]    |
+-----+-----+-----+
| "player100" | ["player125", "player101"] | [42, 42] |
+-----+-----+-----+
| "player102" | ["player101"] | [33]    |
+-----+-----+-----+
...
.

# The following example groups the outputs and restricts the number of rows of the outputs.
nebula> $a = GO FROM "player100" \
    OVER follow YIELD follow._src AS src, follow._dst AS dst; \
    GO 2 STEPS FROM $a.dst OVER follow \
    YIELD $a._src AS src, $a._dst, follow._src, follow._dst |\
    ORDER BY $._src |\
    OFFSET 1 LIMIT 2;
+-----+-----+-----+-----+
| src      | $a._dst      | follow._src | follow._dst |
+-----+-----+-----+-----+
| "player100" | "player125" | "player100" | "player101" |
+-----+-----+-----+-----+
| "player100" | "player101" | "player100" | "player125" |
+-----+-----+-----+-----+
...
.

# The following example determines if $$._player.name IS NOT EMPTY.
nebula> GO FROM "player100" \
    OVER * \
    WHERE $$._player.name IS NOT EMPTY \
    YIELD follow._dst;
+-----+
| follow._dst |
+-----+
| "player125" |
+-----+
...
.

```

```
| "player101" |  
+-----+  
.....
```

Last update: August 3, 2021

4.6.4 FETCH

The `FETCH` statement retrieves the properties of the specified vertices or edges.

OpenCypher Compatibility

This topic applies to native nGQL only.

Fetch vertex properties

SYNTAX

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
[YIELD <output>]
```

Parameter	Description
<code>tag_name</code>	The name of the tag.
<code>*</code>	Represents all the tags in the current graph space.
<code>vid</code>	The vertex ID.
<code>output</code>	Specifies the information to be returned. For more information, see YIELD . If there is no <code>YIELD</code> clause, <code>FETCH</code> returns all the matched information.

FETCH VERTEX PROPERTIES BY ONE TAG

Specify a tag in the `FETCH` statement to fetch the vertex properties by that tag.

```
nebula> FETCH PROP ON player "player100";
+-----+
| vertices_
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+
```

FETCH SPECIFIC PROPERTIES OF A VERTEX

Use a `YIELD` clause to specify the properties to be returned.

```
nebula> FETCH PROP ON player "player100" \
    YIELD player.name;
+-----+
| VertexID      | player.name |
+-----+
| "player100"    | "Tim Duncan" |
+-----+
```

FETCH PROPERTIES OF MULTIPLE VERTICES

Specify multiple VIDs (vertex IDs) to fetch properties of multiple vertices. Separate the VIDs with commas.

```
nebula> FETCH PROP ON player "player101", "player102", "player103";
+-----+
| vertices_
+-----+
| ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----+
| ("player103" :player{age: 32, name: "Rudy Gay"}) |
+-----+
```

FETCH VERTEX PROPERTIES BY MULTIPLE TAGS

Specify multiple tags in the `FETCH` statement to fetch the vertex properties by the tags. Separate the tags with commas.

```
# The following example creates a new tag t1.
nebula> CREATE TAG t1(a string, b int);
```

```
# The following example attaches t1 to the vertex "player100".
nebula> INSERT VERTEX t1(a, b) VALUE "player100":("Hello", 100);

# The following example fetches the properties of vertex "player100" by the tags player and t1.
nebula> FETCH PROP ON player, t1 "player100";
+-----+
| vertices_
+-----+
| ("player100" :t1{a: "Hello", b: 100} :player{age: 42, name: "Tim Duncan"}) |
+-----+
```

You can combine multiple tags with multiple VIDs in a `FETCH` statement.

```
nebula> FETCH PROP ON player, t1 "player100", "player103";
+-----+
| vertices_
+-----+
| ("player100" :t1{a: "Hello", b: 100} :player{age: 42, name: "Tim Duncan"}) |
+-----+
| ("player103" :player{age: 32, name: "Rudy Gay"}) |
+-----+
```

FETCH VERTEX PROPERTIES BY ALL TAGS

Set an asterisk symbol `*` to fetch properties by all tags in the current graph space.

```
nebula> FETCH PROP ON * "player100", "player106", "team200";
+-----+
| vertices_
+-----+
| ("player106" :player{age: 25, name: "Kyle Anderson"}) |
+-----+
| ("team200" :team{name: "Warriors"}) |
+-----+
| ("player100" :t1{a: "Hello", b: 100} :player{age: 42, name: "Tim Duncan"}) |
+-----+
```

Fetch edge properties

SYNTAX

```
FETCH PROP ON <edge_type> <src_vid> -> <dst_vid>[@<rank>] [, <src_vid> -> <dst_vid> ...]
[YIELD <output>]
```

Parameter	Description
<code>edge_type</code>	The name of the edge type.
<code>src_vid</code>	The VID of the source vertex. It specifies the start of an edge.
<code>dst_vid</code>	The VID of the destination vertex. It specifies the end of an edge.
<code>rank</code>	The rank of the edge. It is optional and defaults to <code>0</code> . It distinguishes an edge from other edges with the same edge type, source vertex, destination vertex, and rank.
<code>output</code>	Specifies the information to be returned. For more information, see <code>YIELD</code> . If there is no <code>YIELD</code> clause, <code>FETCH</code> returns all the matched information.

FETCH ALL PROPERTIES OF AN EDGE

The following statement fetches all the properties of the `serve` edge that connects vertex `"player100"` and vertex `"team204"`.

```
nebula> FETCH PROP ON serve "player100" -> "team204";
+-----+
| edges_
+-----+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
+-----+
```

FETCH SPECIFIC PROPERTIES OF AN EDGE

Use a `YIELD` clause to fetch specific properties of an edge.

```
nebula> FETCH PROP ON serve "player100" -> "team204" \
    YIELD serve.start_year;
+-----+
```

```

| serve._src | serve._dst | serve._rank | serve.start_year |
+-----+-----+-----+-----+
| "player100" | "team204" | 0 | 1997 |
+-----+-----+-----+

```

FETCH PROPERTIES OF MULTIPLE EDGES

Specify multiple edge patterns (`<src_vid> -> <dst_vid>[@<rank>]`) to fetch properties of multiple edges. Separate the edge patterns with commas.

```

nebula> FETCH PROP ON serve "player100" -> "team204", "player133" -> "team202";
+-----+
| edges_
+-----+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
+-----+
| [:serve "player133"->"team202" @0 {end_year: 2011, start_year: 2002}] |
+-----+

```

Fetch properties based on edge rank

If there are multiple edges with the same edge type, source vertex, and destination vertex, you can specify the rank to fetch the properties on the correct edge.

```

# The following example inserts edges with different ranks and property values.
nebula> insert edge serve(start_year,end_year) \
  values "player100"->"team204"@1:(1998, 2017);

nebula> insert edge serve(start_year,end_year) \
  values "player100"->"team204"@2:(1990, 2018);

# By default, the FETCH statement returns the edge whose rank is 0.
nebula> FETCH PROP ON serve "player100" -> "team204";
+-----+
| edges_
+-----+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
+-----+

# To fetch on an edge whose rank is not 0, set its rank in the FETCH statement.
nebula> FETCH PROP ON serve "player100" -> "team204"@1;
+-----+
| edges_
+-----+
| [:serve "player100"->"team204" @1 {end_year: 2017, start_year: 1998}] |
+-----+

```

Use FETCH in composite queries

A common way to use `FETCH` is to combine it with native nGQL such as `GO`.

The following statement returns the `degree` values of the `follow` edges that start from vertex `"player101"`.

```

nebula> GO FROM "player101" OVER follow \
  YIELD follow._src AS s, follow._dst AS d | \
  FETCH PROP ON follow $-.s -> $-.d \
  YIELD follow.degree;
+-----+-----+-----+
| follow._src | follow._dst | follow._rank | follow.degree |
+-----+-----+-----+
| "player101" | "player100" | 0 | 95 |
+-----+-----+-----+
| "player101" | "player102" | 0 | 90 |
+-----+-----+-----+
| "player101" | "player125" | 0 | 95 |
+-----+-----+-----+

```

Or you can use user-defined variables to construct similar queries.

```

nebula> $var = GO FROM "player101" OVER follow \
  YIELD follow._src AS s, follow._dst AS d; \
  FETCH PROP ON follow $var.s -> $var.d \
  YIELD follow.degree;
+-----+-----+-----+
| follow._src | follow._dst | follow._rank | follow.degree |
+-----+-----+-----+
| "player101" | "player100" | 0 | 95 |
+-----+-----+-----+
| "player101" | "player102" | 0 | 90 |
+-----+-----+-----+

```

"player101" "player125" 0	95	
+-----+-----+-----+		

For more information about composite queries, see [Composite queries \(clause structure\)](#).

Last update: August 3, 2021

4.6.5 UNWIND

The `UNWIND` statement splits a list into separated rows.

`UNWIND` can function as an individual statement or a clause in a statement.

Syntax

```
UNWIND <list> AS <alias> <RETURN clause>
```

Split a list

The following example splits the list `[1, 2, 3]` into three rows.

```
nebula> UNWIND [1,2,3] AS n RETURN n;
+---+
| n |
+---+
| 1 |
+---+
| 2 |
+---+
| 3 |
+---+
```

Return a list with distinct items

Use `WITH DISTINCT` in the `UNWIND` statement to return a list with distinct items.

EXAMPLE 1

The following statement:

1. Splits the list `[1, 1, 2, 2, 3, 3]` into rows.
2. Removes duplicated rows.
3. Sorts the rows.
4. Transforms the rows to a list.

```
nebula> WITH [1,1,2,2,3,3] AS n \
    UNWIND n AS r \
    WITH DISTINCT r AS r \
    ORDER BY r \
    RETURN collect(r);
+-----+
| collect(r) |
+-----+
| [1, 2, 3] |
+-----+
```

Example 2

The following statement:

1. Outputs the vertices on the matched path into a list.
2. Splits the list into rows.
3. Removes duplicated rows.
4. Transforms the rows to a list.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})--(v2) \
    WITH nodes(p) AS n \
    UNWIND n AS r \
    WITH DISTINCT r AS r \
    RETURN collect(r);
+-----+
| collect(r) |
+-----+
```

```
| [{"player100": player{age: 42, name: "Tim Duncan"}}, {"player101": player{age: 36, name: "Tony Parker"}}, {"team204": {teamName: "Spurs")}, {"player102": player{age: 33, name: "LaMarcus Aldridge"}}, {"player125": player{age: 41, name: "Manu Ginobili")}, {"player104": player{age: 32, name: "Marco Belinelli"}}, {"player144": player{age: 47, name: "Shaqiue O'Neal"}}, {"player105": player{age: 31, name: "Danny Green"}}, {"player113": player{age: 29, name: "Dejounte Murray"}}, {"player107": player{age: 32, name: "Aron Baynes"}}, {"player109": player{age: 34, name: "Tiago Splitter")}, {"player108": player{age: 36, name: "Boris Diaw"}]}] |  
+-----+  
|
```

Last update: August 2, 2021

4.6.6 SHOW

SHOW CHARSET

The `SHOW CHARSET` statement shows the available character sets.

Currently available types are `utf8` and `utf8mb4`. The default charset type is `utf8`. Nebula Graph extends the `utf8` to support four-byte characters. Therefore `utf8` and `utf8mb4` are equivalent.

SYNTAX

```
SHOW CHARSET;
```

EXAMPLE

```
nebula> SHOW CHARSET;
+-----+-----+-----+
| Charset | Description | Default collation | Maxlen |
+-----+-----+-----+
| "utf8" | "UTF-8 Unicode" | "utf8_bin" | 4 |
+-----+-----+-----+
```

Parameter	Description
Charset	The name of the character set.
Description	The description of the character set.
Default collation	The default collation of the character set.
Maxlen	The maximum number of bytes required to store one character.

Last update: July 27, 2021

SHOW COLLATION

The `SHOW COLLATION` statement shows 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 `utf8mb4_bin` and `utf8mb4_general_ci` are case-insensitive.

SYNTAX

```
SHOW COLLATION;
```

EXAMPLE

```
nebula> SHOW COLLATION;
+-----+-----+
| Collation | Charset |
+-----+-----+
| "utf8_bin" | "utf8" |
+-----+-----+
```

Parameter	Description
<code>Collation</code>	The name of the collation.
<code>Charset</code>	The name of the character set with which the collation is associated.

Last update: July 27, 2021

SHOW CREATE SPACE

The `SHOW CREATE SPACE` statement shows the creating statement of the specified graph space.

For details about the graph space information, see [CREATE SPACE](#).

SYNTAX

```
SHOW CREATE SPACE <space_name>;
```

EXAMPLE

```
nebula> SHOW CREATE SPACE basketballplayer;
+-----+
| Space          | Create
Space
+-----+
| "basketballplayer" | "CREATE SPACE `basketballplayer` (partition_num = 10, replica_factor = 1, charset = utf8, collate = utf8_bin, vid_type =
FIXED_STRING(32))" |
+-----+
+-----+
```

Last update: July 27, 2021

SHOW CREATE TAG/EDGE

The `SHOW CREATE TAG` statement shows the basic information of the specified tag. For details about the tag, see [CREATE TAG](#).

The `SHOW CREATE EDGE` statement shows the basic information of the specified edge type. For details about the edge type, see [CREATE EDGE](#).

SYNTAX

```
SHOW CREATE {TAG <tag_name> | EDGE <edge_name>};
```

EXAMPLES

```
nebula> SHOW CREATE TAG player;
+-----+-----+
| Tag   | Create Tag
+-----+-----+
| "player" | "CREATE TAG `player` (
|           |   `name` string NULL,
|           |   `age` int64 NULL
|           | ) ttl_duration = 0, ttl_col = "" |
+-----+-----+

nebula> SHOW CREATE EDGE follow;
+-----+-----+
| Edge  | Create Edge
+-----+-----+
| "follow" | "CREATE EDGE `follow` (
|           |   `degree` int64 NULL
|           | ) ttl_duration = 0, ttl_col = "" |
+-----+-----+
```

Last update: August 4, 2021

SHOW HOSTS

The `SHOW HOSTS` statement shows the host and version information of Graph Service, Storage Service, and Meta Service.

SYNTAX

```
SHOW HOSTS [GRAPH | STORAGE | META];
```

If you return `SHOW HOSTS` without the service name, it will show the host information of Storage Service, as well as the leader number, leader distribution, and partition distribution.

EXAMPLES

Last update: August 4, 2021

SHOW INDEX STATUS

The `SHOW INDEX STATUS` statement shows the status of jobs that rebuild native indexes, which helps check whether a native index is successfully rebuilt or not.

SYNTAX

```
SHOW {TAG | EDGE} INDEX STATUS;
```

EXAMPLES

```
nebula> SHOW TAG INDEX STATUS;
+-----+-----+
| Name      | Index Status |
+-----+-----+
| "like_index_0" | "FINISHED"  |
+-----+-----+
| "like1"      | "FINISHED"  |
+-----+-----+  
nebula> SHOW EDGE INDEX STATUS;
+-----+-----+
| Name      | Index Status |
+-----+-----+
| "index_follow" | "FINISHED"  |
+-----+-----+
```

RELATED TOPICS

- [Job manager and the JOB statements](#)
- [REBUILD NATIVE INDEX](#)

Last update: July 27, 2021

SHOW INDEXES

The `SHOW INDEXES` statement shows the names of existing native indexes.

SYNTAX

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

EXAMPLES

```
nebula> SHOW TAG INDEXES;
+-----+
| Names      |
+-----+
| "play_age_0"      |
+-----+
| "player_index_0"  |
+-----+  
  
nebula> SHOW EDGE INDEXES;
+-----+
| Names      |
+-----+
| "index_follow"  |
+-----+
```

Last update: July 27, 2021

SHOW PARTS

The `SHOW PARTS` statement shows the information of a specified partition or all partitions in a graph space.

SYNTAX

```
SHOW PARTS [<part_id>];
```

EXAMPLES

```
nebula> SHOW PARTS;
+-----+-----+-----+
| Partition ID | Leader | Peers | Losts |
+-----+-----+-----+
| 1 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
+-----+-----+-----+
| 2 | "192.168.2.2:9779" | "192.168.2.2:9779" | "" |
+-----+-----+-----+
| 3 | "192.168.2.3:9779" | "192.168.2.3:9779" | "" |
+-----+-----+-----+
| 4 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
+-----+-----+-----+
| 5 | "192.168.2.2:9779" | "192.168.2.2:9779" | "" |
+-----+-----+-----+
| 6 | "192.168.2.3:9779" | "192.168.2.3:9779" | "" |
+-----+-----+-----+
| 7 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
+-----+-----+-----+
| 8 | "192.168.2.2:9779" | "192.168.2.2:9779" | "" |
+-----+-----+-----+
| 9 | "192.168.2.3:9779" | "192.168.2.3:9779" | "" |
+-----+-----+-----+
| 10 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
+-----+-----+-----+  
  
nebula> SHOW PARTS 1;
+-----+-----+-----+
| Partition ID | Leader | Peers | Losts |
+-----+-----+-----+
| 1 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
+-----+-----+-----+
```

The descriptions are as follows.

Parameter	Description
Partition ID	The ID of the partition.
Leader	The IP address and the port of the leader.
Peers	The IP addresses and the ports of all the replicas.
Losts	The IP addresses and the ports of replicas at fault.

Last update: August 23, 2021

SHOW ROLES

The `SHOW ROLES` statement shows the roles that are assigned to a user account.

The return message differs according to the role of the user who is running this statement:

- If the user is a `GOD` or `ADMIN` and is granted access to the specified graph space, Nebula Graph shows all roles in this graph space except for `GOD`.
- If the user is a `DBA`, `USER`, or `GUEST` and is granted access to the specified graph space, Nebula Graph shows the user's own role in this graph space.
- If the user does not have access to the specified graph space, Nebula Graph returns `PermissionError`.

For more information about roles, see [Roles and privileges](#).

SYNTAX

```
SHOW ROLES IN <space_name>;
```

EXAMPLE

```
nebula> SHOW ROLES in basketballplayer;
+-----+-----+
| Account | Role Type |
+-----+-----+
| "user1" | "ADMIN"   |
+-----+-----+
```

Last update: July 27, 2021

SHOW SNAPSHOTS

The `SHOW SNAPSHOTS` statement shows the information of all the snapshots.

For how to create a snapshot and backup data, see [Snapshot](#).

ROLE REQUIREMENT

Only the `root` user who has the `GOD` role can use the `SHOW SNAPSHOTS` statement.

SYNTAX

```
SHOW SNAPSHOTS;
```

EXAMPLE

```
nebula> SHOW SNAPSHOTS;
+-----+-----+-----+
| Name      | Status | Hosts
+-----+-----+-----+
| "SNAPSHOT_2020_12_16_11_13_55" | "VALID" | "storaged0:9779, storaged1:9779, storaged2:9779" |
+-----+-----+-----+
| "SNAPSHOT_2020_12_16_11_14_10" | "VALID" | "storaged0:9779, storaged1:9779, storaged2:9779" |
+-----+-----+-----+
```

Last update: July 27, 2021

SHOW SPACES

The `SHOW SPACES` statement shows existing graph spaces in Nebula Graph.

For how to create a graph space, see [CREATE SPACE](#).

SYNTAX

```
SHOW SPACES;
```

EXAMPLE

```
nebula> SHOW SPACES;
+-----+
| Name      |
+-----+
| "docs"    |
+-----+
| "basketballplayer" |
+-----+
```

Last update: July 27, 2021

SHOW STATS

The `SHOW STATS` statement shows the statistics of the graph space collected by the latest `STATS` job.

The statistics include the following information:

- The number of vertices in the graph space
- The number of edges in the graph space
- The number of vertices of each tag
- The number of edges of each edge type

PREREQUISITES

You have to run the `SUBMIT JOB STATS` statement in the graph space where you want to collect statistics. For more information, see [SUBMIT JOB STATS](#).

⚠ Caution

The result of the `SHOW STATS` statement is based on the last executed `SUBMIT JOB STATS` statement. If you want to update the result, run `SUBMIT JOB STATS` again. Otherwise the statistics will be wrong.

SYNTAX

```
SHOW STATS;
```

EXAMPLES

```
# Choose a graph space.
nebula> USE basketballplayer;

# Start SUBMIT JOB STATS.
nebula> SUBMIT JOB STATS;
+-----+
| New Job Id |
+-----+
| 98          |
+-----+

# Make sure the job executes successfully.
nebula> SHOW JOB 98;
+-----+-----+-----+-----+-----+
| Job Id(TaskId) | Command(Dest) | Status      | Start Time | Stop Time  |
+-----+-----+-----+-----+-----+
| 98           | "STATS"      | "FINISHED"  | 1606552675 | 1606552675 |
+-----+-----+-----+-----+-----+
| 0            | "storaged2"  | "FINISHED"  | 1606552675 | 1606552675 |
+-----+-----+-----+-----+-----+
| 1            | "storaged0"  | "FINISHED"  | 1606552675 | 1606552675 |
+-----+-----+-----+-----+-----+
| 2            | "storaged1"  | "FINISHED"  | 1606552675 | 1606552675 |
+-----+-----+-----+-----+-----+

# Show the statistics of the graph space.
nebula> SHOW STATS;
+-----+-----+
| Type      | Name      | Count |
+-----+-----+
| "Tag"     | "player"  | 51    |
+-----+-----+
| "Tag"     | "team"    | 30    |
+-----+-----+
| "Edge"    | "like"    | 81    |
+-----+-----+
| "Edge"    | "serve"   | 152   |
+-----+-----+
| "Space"   | "vertices" | 81    |
+-----+-----+
| "Space"   | "edges"   | 233   |
+-----+-----+
```

Last update: July 27, 2021

SHOW TAGS/EDGES

The `SHOW TAGS` statement shows all the tags in the current graph space.

The `SHOW EDGES` statement shows all the edge types in the current graph space.

SYNTAX

```
SHOW {TAGS | EDGES};
```

EXAMPLES

```
nebula> SHOW TAGS;
+-----+
| Name   |
+-----+
| "player" |
+-----+
| "star"   |
+-----+
| "team"   |
+-----+  
  
nebula> SHOW EDGES;
+-----+
| Name   |
+-----+
| "like"  |
+-----+
| "serve" |
+-----+
```

Last update: July 27, 2021

SHOW USERS

The `SHOW USERS` statement shows the user information.

ROLE REQUIREMENT

Only the `root` user who has the `GOD` role can use the `SHOW USERS` statement.

SYNTAX

```
SHOW USERS;
```

EXAMPLE

```
nebula> SHOW USERS;
+-----+
| Account |
+-----+
| "root"  |
+-----+
| "user1" |
+-----+
```

Last update: July 27, 2021

SHOW SESSIONS

The `SHOW SESSIONS` statement shows the information of all the sessions. It can also show a specified session with its ID.

PRECAUTIONS

When you log in to the database using Nebula Console, a session will be created. The client will execute the API `release` to release the session and clear the session information when you run `exit` after the operation ends.

If you exit the database in unexpected ways with the `session_idle_timeout_secs` in `nebula-graphd.conf` undetermined, the session will not be released automatically.

For those sessions that are not automatically released, you need to delete them manually (TODO: coding).

SYNTAX

```
SHOW SESSIONS;  
SHOW SESSION <Session_Id>;
```

EXAMPLES

Parameter	Description
SessionId	The session ID, namely the identifier of a session.
UserName	The username in a session.
SpaceName	The name of the graph space that the user uses currently. It is null ("") when you first log in because there is no specified graph space.
CreateTime	The time when the session is created, namely the time when the user logs in. The time zone is specified by <code>timezone_name</code> in the configuration file.
UpdateTime	The system will update the time when there is an operation. The time zone is specified by <code>timezone_name</code> in the configuration file.
GraphAddr	The IP address and port of the Graph server that hosts the session.
Timezone	A reserved parameter that has no specified meaning for now.
ClientIp	The IP address of the client.

SHOW QUERIES

The `SHOW QUERIES` statement shows the information of working queries in the current session.

Note

To terminate queries, see [Kill Query](#).

PRECAUTIONS

- The `SHOW QUERIES` statement gets the status of queries in the current session from the local cache with almost no latency.
- The `SHOW ALL QUERIES` statement gets the information of queries in all the sessions from the Meta Service. The information will be synchronized to the Meta Service according to the interval defined by `session_reclaim_interval_secs`. Therefore the information that you get from the client may belong to the last synchronization interval.

SYNTAX

```
SHOW [ALL] QUERIES;
```

EXAMPLES

```
nebula> SHOW QUERIES;
+-----+-----+-----+-----+-----+-----+-----+
| SessionID | ExecutionPlanID | User | Host | StartTime | DurationInUsec | Status | Query |
+-----+-----+-----+-----+-----+-----+-----+
| 1625463842921750 | 46 | "root" | "192.168.x.x":9669 | 2021-07-05T05:44:19.502903 | 0 | "RUNNING" | "SHOW QUERIES;" |
+-----+-----+-----+-----+-----+-----+-----+
nebula> SHOW ALL QUERIES;
+-----+-----+-----+-----+-----+-----+
| SessionID | ExecutionPlanID | User | Host | StartTime | DurationInUsec | Status | |
Query | | | | | |
+-----+-----+-----+-----+-----+-----+
| 1625456037718757 | 54 | "user1" | "192.168.x.x":9669 | 2021-07-05T05:51:08.691318 | 1504502 | "RUNNING" | "MATCH p=(v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
+-----+-----+-----+-----+-----+-----+
# The following statement returns the top 10 queries that have the longest duration.
nebula> SHOW ALL QUERIES | ORDER BY $-.DurationInUsec DESC | LIMIT 10;
+-----+-----+-----+-----+-----+-----+
| SessionID | ExecutionPlanID | User | Host | StartTime | DurationInUsec | Status | |
Query | | | | | |
+-----+-----+-----+-----+-----+-----+
| 1625471375320831 | 98 | "user2" | "192.168.x.x":9669 | 2021-07-05T07:50:24.461779 | 2608176 | "RUNNING" | "MATCH (v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
+-----+-----+-----+-----+-----+-----+
| 1625456037718757 | 99 | "user1" | "192.168.x.x":9669 | 2021-07-05T07:50:24.910616 | 2159333 | "RUNNING" | "MATCH (v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
+-----+-----+-----+-----+-----+-----+
```

The descriptions are as follows.

Parameter	Description
SessionID	The session ID.
ExecutionPlanID	The ID of the execution plan.
User	The username that executes the query.
Host	The IP address and port of the Graph server that hosts the session.
StartTime	The time when the query starts.
DurationInusec	The duration of the query. The unit is microsecond.
Status	The current status of the query.
Query	The query statement.

Last update: July 27, 2021

4.7 Clauses and options

4.7.1 GROUP BY

The `GROUP BY` clause can be used to aggregate data.

OpenCypher Compatibility

This topic applies to native nGQL only.

You can also use the `count()` function to aggregate data.

```
nebula> MATCH (v:player)<-[:follow]-(:player) RETURN v.name AS Name, count(*) as cnt ORDER BY cnt DESC;
+-----+-----+
| Name | cnt |
+-----+-----+
| "Tim Duncan" | 10 |
+-----+-----+
| "LeBron James" | 6 |
+-----+-----+
| "Tony Parker" | 5 |
+-----+-----+
| "Chris Paul" | 4 |
+-----+-----+
| "Manu Ginobili" | 4 |
+-----+-----+
...
```

Syntax

The `GROUP BY` clause groups the rows with the same value. Then operations such as counting, sorting, and calculation can be applied.

The `GROUP BY` clause works after the pipe symbol (`|`) and before a `YIELD` clause.

```
| GROUP BY <var> YIELD <var>, <aggregation_function(var)>
```

The `aggregation_function()` function supports `avg()`, `sum()`, `max()`, `min()`, `count()`, `collect()`, and `std()`.

Examples

The following statement finds all the vertices connected directly to vertex `"player100"`, groups the result set by player names, and counts how many times the name shows up in the result set.

```
nebula> GO FROM "player100" OVER follow BIDIRECT \
    YIELD $$.player.name as Name \
    | GROUP BY $-.Name \
    YIELD $-.Name as Player, count(*) AS Name_Count;
+-----+-----+
| Player | Name_Count |
+-----+-----+
| "Tiago Splitter" | 1 |
+-----+-----+
| "Aron Baynes" | 1 |
+-----+-----+
| "Boris Diaw" | 1 |
+-----+-----+
| "Manu Ginobili" | 2 |
+-----+-----+
| "Dejounte Murray" | 1 |
+-----+-----+
| "Danny Green" | 1 |
+-----+-----+
| "Tony Parker" | 2 |
+-----+-----+
| "Shaquille O'Neal" | 1 |
+-----+-----+
| "LaMarcus Aldridge" | 1 |
+-----+-----+
| "Marco Belinelli" | 1 |
+-----+-----+
```

Group and calculate with functions

The following statement finds all the vertices connected directly to vertex "player100" , groups the result set by source vertices, and returns the sum of degree values.

```
nebula> GO FROM "player100" OVER follow \
    YIELD follow_src AS player, follow.degree AS degree \
    | GROUP BY $-.player \
    YIELD sum($-.degree);
+-----+
| sum($-.degree) |
+-----+
| 190           |
+-----+
```

For more information about the `sum()` function, see [Built-in math functions](#).

Last update: August 23, 2021

4.7.2 LIMIT AND SKIP

The `LIMIT` clause constrains the number of rows in the output.

- Native nGQL: A pipe `|` must be used. And an offset can be ignored.
- OpenCypher style: No pipes are permitted. And you can use `SKIP` to indicate an offset.

Note

When using `LIMIT` in either syntax above, 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.

Native nGQL syntax

In native nGQL, `LIMIT` works the same as in `SQL`, and must be used with pipe `|`. The `LIMIT` clause accepts one or two parameters. The values of both arguments must be non-negative integers.

```
YIELD <var>
[| LIMIT [<offset_value>,<number_rows>];
```

Parameter	Description
<code>var</code>	The columns or calculations that you wish to sort.
<code>offset_value</code>	The offset value. It defines from which row to start returning. The offset starts from <code>0</code> . The default value is <code>0</code> , which returns from the first row.
<code>number_rows</code>	It constrains the total number of returned rows.

EXAMPLES

```
# The following example returns the 3 rows of data starting from the second row of the sorted output.
nebula> GO FROM "player100" OVER follow REVERSELY \
    YIELD $$.player.name AS Friend, $$.player.age AS Age \
    | ORDER BY Age, Friend \
    | LIMIT 1, 3;
+-----+-----+
| Friend | Age |
+-----+-----+
| "Danny Green" | 31 |
+-----+-----+
| "Aron Baynes" | 32 |
+-----+-----+
| "Marco Belinelli" | 32 |
+-----+-----+
```

OpenCypher syntax

```
RETURN <var>
[SKIP <offset>]
[LIMIT <number_rows>];
```

Parameter	Description
<code>var</code>	The columns or calculations that you wish to sort.
<code>offset</code>	The offset value. It defines from which row to start returning. The offset starts from <code>0</code> . The default value is <code>0</code> , which returns from the first row.
<code>number_rows</code>	It constrains the total number of returned rows.

Both `offset` and `number_rows` accept expressions, but the result of the expression must be a non-negative integer.

Note

Fraction expressions composed of two integers are automatically floored to integers. For example, `8/6` is floored to 1.

EXAMPLES

```
nebula> MATCH (v:player) RETURN v.name AS Name, v.age AS Age \
    ORDER BY Age LIMIT 5;
+-----+-----+
| Name | Age |
+-----+-----+
| "Luka Doncic" | 20 |
+-----+-----+
| "Ben Simmons" | 22 |
+-----+-----+
| "Kristaps Porzingis" | 23 |
+-----+-----+
| "Giannis Antetokounmpo" | 24 |
+-----+-----+
| "Kyle Anderson" | 25 |
+-----+-----+  
  
nebula> MATCH (v:player) RETURN v.name AS Name, v.age AS Age \
    ORDER BY Age LIMIT rand32(5);
+-----+-----+
| Name | Age |
+-----+-----+
| "Luka Doncic" | 20 |
+-----+-----+
| "Ben Simmons" | 22 |
+-----+-----+
| "Kristaps Porzingis" | 23 |
+-----+-----+
| "Giannis Antetokounmpo" | 24 |
+-----+-----+
```

EXAMPLES OF SKIP

You can use `SKIP <offset>` to skip the top N rows of the output and return the rest of the output. So, there is no need to add `LIMIT <number_rows>`.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.name AS Name, v2.age AS Age \
    ORDER BY Age DESC SKIP 1;
+-----+-----+
| Name | Age |
+-----+-----+
| "Manu Ginobili" | 41 |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+  
  
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.name AS Name, v2.age AS Age \
    ORDER BY Age DESC SKIP 1+1;
+-----+-----+
| Name | Age |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
```

You can use `SKIP <offset>` and `LIMIT <number_rows>` together to return the data of the middle N rows.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.name AS Name, v2.age AS Age \
    ORDER BY Age DESC SKIP 1 LIMIT 1;
+-----+-----+
| Name | Age |
+-----+-----+
| "Manu Ginobili" | 41 |
+-----+-----+
```

Last update: August 23, 2021

4.7.3 ORDER BY

The `ORDER BY` clause specifies the order of the rows in the output.

- Native nGQL: You must use a pipe (|) and an `ORDER BY` clause after `YIELD` clause.
- OpenCypher style: No pipes are permitted. The `ORDER BY` clause follows a `RETURN` clause.

There are two order options:

- `ASC` : Ascending. `ASC` is the default order.
- `DESC` : Descending.

Native nGQL Syntax

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

EXAMPLES

```
nebula> FETCH PROP ON player "player100", "player101", "player102", "player103" \
    YIELD player.age AS age, player.name AS name \
    | ORDER BY age ASC, name DESC;
+-----+-----+
| VertexID | age | name
+-----+-----+
| "player103" | 32 | "Rudy Gay"
+-----+-----+
| "player102" | 33 | "LaMarcus Aldridge"
+-----+-----+
| "player101" | 36 | "Tony Parker"
+-----+-----+
| "player100" | 42 | "Tim Duncan"
+-----+-----+
```

OpenCypher Syntax

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

EXAMPLES

```
nebula> MATCH (v:player) RETURN v.name AS Name, v.age AS Age \
    ORDER BY Name DESC;
+-----+-----+
| Name | Age |
+-----+-----+
| "Yao Ming" | 38 |
+-----+-----+
| "Vince Carter" | 42 |
+-----+-----+
| "Tracy McGrady" | 39 |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
| "Tim Duncan" | 42 |
+-----+-----+
...
# In the following example, nGQL sorts the rows by age first. If multiple people are of the same age, nGQL will then sort them by name.
nebula> MATCH (v:player) RETURN v.age AS Age, v.name AS Name \
    ORDER BY Age DESC, Name ASC;
+-----+-----+
| Age | Name |
+-----+-----+
| 47 | "Shaquille O'Neal" |
+-----+-----+
| 46 | "Grant Hill" |
+-----+-----+
| 45 | "Jason Kidd" |
+-----+-----+
| 45 | "Steve Nash" |
+-----+-----+
...
```

Order of NULL values

nGQL lists NULL values at the end of the output for ascending sorting, and at the start for descending sorting.

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.name AS Name, v2.age AS Age \
    ORDER BY Age;
+-----+-----+
| Name | Age |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
| "Manu Ginobili" | 41 |
+-----+-----+
| "Spurs" | __NULL__ |
+-----+-----+  
  
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.name AS Name, v2.age AS Age \
    ORDER BY Age DESC;
+-----+-----+
| Name | Age |
+-----+-----+
| "Spurs" | __NULL__ |
+-----+-----+
| "Manu Ginobili" | 41 |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
```

Last update: August 6, 2021

4.7.4 RETURN

The `RETURN` clause defines the output of an nGQL query. To return multiple fields, separate them with commas.

`RETURN` can lead a clause or a statement:

- A `RETURN` clause can work in openCypher statements in nGQL, such as `MATCH` or `UNWIND`.
- A `RETURN` statement can work independently to output the result of an expression.

OpenCypher compatibility

This topic applies to the openCypher syntax in nGQL only. For native nGQL, use `YIELD`.

`RETURN` does not support the following openCypher features yet.

- Return variables with uncommon characters, for example:

```
MATCH (`non-english_characters` :player) \
RETURN `non-english_characters`;
```

- Set a pattern in the `RETURN` clause and return all elements that this pattern matches, for example:

```
MATCH (v:player) \
RETURN (v)-[e]->(v2);
```

Legacy version compatibility

- In nGQL 1.x, `RETURN` works with native nGQL with the `RETURN <var_ref> IF <var_ref> IS NOT NULL` syntax.
- In nGQL 2.0, `RETURN` does not work with native nGQL.

Return vertices

```
nebula> MATCH (v:player) \
    RETURN v;
+-----+
| v
+-----+
| ("player104" :player{age: 32, name: "Marco Belinelli"})
+-----+
| ("player107" :player{age: 32, name: "Aron Baynes"})
+-----+
| ("player116" :player{age: 34, name: "LeBron James"})
+-----+
| ("player120" :player{age: 29, name: "James Harden"})
+-----+
| ("player125" :player{age: 41, name: "Manu Ginobili"})
+-----+
...
```

Return edges

```
nebula> MATCH (v:player)-[e]->() \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player104"->"player100" @0 {degree: 55}]
+-----+
| [:follow "player104"->"player101" @0 {degree: 50}]
+-----+
| [:follow "player104"->"player105" @0 {degree: 60}]
+-----+
| [:serve "player104"->"team200" @0 {end_year: 2009, start_year: 2007}]
+-----+
| [:serve "player104"->"team208" @0 {end_year: 2016, start_year: 2015}]
+-----+
...
```

Return properties

To return a vertex or edge property, use the `{<vertex_name>|<edge_name>}.<property>` syntax.

```
nebula> MATCH (v:player) \
    RETURN v.name, v.age \
    LIMIT 3;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Rajon Rondo" | 33 |
+-----+-----+
| "Rudy Gay" | 32 |
+-----+-----+
| "Dejounte Murray" | 29 |
+-----+-----+
```

Return all elements

To return all the elements that this pattern matches, use an asterisk (*).

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN *;
+-----+
| v |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
    RETURN *;
+-----+
| v |
+-----+
| v2 |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:follow "player100"->"player101" @0 {degree: 95}] | ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:follow "player100"->"player125" @0 {degree: 95}] | ("player125" :player{age: 41, name: "Manu Ginobili"}) |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] | ("team204" :team{name: "Spurs"}) |
+-----+
```

Rename a field

Use the `AS <alias>` syntax to rename a field in the output.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[:serve]->(v2) \
    RETURN v2.name AS Team;
+-----+
| Team |
+-----+
| "Spurs" |
+-----+
nebula> RETURN "Amber" AS Name;
+-----+
| Name |
+-----+
| "Amber" |
+-----+
```

Return a non-existing property

If a property matched does not exist, `NULL` is returned.

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
    RETURN v2.name, type(e), v2.age;
+-----+-----+
| v2.name | type(e) | v2.age |
+-----+-----+
| "Tony Parker" | "follow" | 36 |
+-----+-----+
| "Manu Ginobili" | "follow" | 41 |
+-----+
```

```
+-----+-----+-----+
| "Spurs"      | "serve"   | __NULL__ |
+-----+-----+-----+
```

Return expression results

To return the results of expressions such as literals, functions, or predicates, set them in a `RETURN` clause.

```
nebula> MATCH (v:player{name:"Tony Parker"})-->(v2:player) \
    RETURN DISTINCT v2.name, "Hello"+" graphs!", v2.age > 35;
+-----+-----+-----+
| v2.name      | (Hello+ graphs!) | (v2.age>35) |
+-----+-----+-----+
| "Tim Duncan" | "Hello graphs!" | true      |
+-----+-----+-----+
| "LaMarcus Aldridge" | "Hello graphs!" | false     |
+-----+-----+-----+
| "Manu Ginobili" | "Hello graphs!" | true      |
+-----+-----+-----+


nebula> RETURN 1+1;
+-----+
| (1+1) |
+-----+
| 2 |
+-----+


nebula> RETURN 3 > 1;
+-----+
| (3>1) |
+-----+
| true |
+-----+


RETURN 1+1, rand32(1, 5);
+-----+-----+
| (1+1) | rand32(1,5) |
+-----+-----+
| 2      | 1           |
+-----+-----+
```

Return unique fields

Use `DISTINCT` to remove duplicate fields in the result set.

```
# Before using DISTINCT.
nebula> MATCH (v:player{name:"Tony Parker"})--(v2:player) \
    RETURN v2.name, v2.age;
+-----+-----+
| v2.name      | v2.age   |
+-----+-----+
| "Tim Duncan" | 42      |
+-----+-----+
| "LaMarcus Aldridge" | 33      |
+-----+-----+
| "Marco Belinelli" | 32      |
+-----+-----+
| "Boris Diaw" | 36      |
+-----+-----+
| "Dejounte Murray" | 29      |
+-----+-----+
| "Tim Duncan" | 42      |
+-----+-----+
| "LaMarcus Aldridge" | 33      |
+-----+-----+
| "Manu Ginobili" | 41      |
+-----+-----+


# After using DISTINCT.
nebula> MATCH (v:player{name:"Tony Parker"})--(v2:player) \
    RETURN DISTINCT v2.name, v2.age;
+-----+-----+
| v2.name      | v2.age   |
+-----+-----+
| "Tim Duncan" | 42      |
+-----+-----+
| "LaMarcus Aldridge" | 33      |
+-----+-----+
| "Marco Belinelli" | 32      |
+-----+-----+
| "Boris Diaw" | 36      |
+-----+-----+
| "Dejounte Murray" | 29      |
+-----+-----+
| "Manu Ginobili" | 41      |
+-----+-----+
```

Last update: August 6, 2021

4.7.5 TTL

TTL (Time To Live) specifies a timeout for a property. Once timed out, the property expires.

OpenCypher Compatibility

This topic applies to native nGQL only.

Precautions

- You CANNOT modify a property schema with TTL options on it.
- TTL options and indexes have coexistence issues.
 - + TTL options and indexes CANNOT coexist on a tag or an edge type. If there is an index on a property, you cannot set TTL options on other properties.
 - + If there are TTL options on a tag, an edge type, or a property, you can still add an index on them.

Data expiration and deletion

VERTEX PROPERTY EXPIRATION

Vertex property expiration has the following impact.

- If a vertex has only one tag, once a property of the vertex expires, the vertex expires.
- If a vertex has multiple tags, once a property of the vertex expires, properties bound to the same tag with the expired property also expire, but the vertex does not expire and other properties of it remain untouched.

EDGE PROPERTY EXPIRATION

Since an edge can have only one edge type, once an edge property expires, the edge expires.

DATA DELETION

The expired data are still stored on the disk, but queries will filter them out.

Nebula Graph automatically deletes the expired data and reclaims the disk space during the next [compaction](#).

Note

If TTL is [disabled](#), the corresponding data deleted after the last compaction can be queried again.

TTL options

The native nGQL TTL feature has the following options.

Option	Description
<code>ttl_col</code>	Specifies the property to set a timeout on. The data type of the property must be <code>int</code> or <code>timestamp</code> .
<code>ttl_duration</code>	Specifies the timeout adds-on value in seconds. The value must be a non-negative <code>int64</code> number. A property expires if the sum of its value and the <code>ttl_duration</code> value is smaller than the current timestamp. If the <code>ttl_duration</code> value is <code>0</code> , the property never expires.

Use TTL options

You must use the TTL options together to set a valid timeout on a property.

SET A TIMEOUT IF A TAG OR AN EDGE TYPE EXISTS

If a tag or an edge type is already created, to set a timeout on a property bound to the tag or edge type, use `ALTER` to update the tag or edge type.

```
# Create a tag.
nebula> CREATE TAG t1 (a timestamp);

# Use ALTER to update the tag and set the TTL options.
nebula> ALTER TAG t1 ttl_col = "a", ttl_duration = 5;

# Insert a vertex with tag t1. The vertex expires 5 seconds after the insertion.
nebula> INSERT VERTEX t1(a) values "101":(now());
```

SET A TIMEOUT WHEN CREATING A TAG OR AN EDGE TYPE

Use TTL options in the `CREATE` statement to set a timeout when creating a tag or an edge type. For more information, see [CREATE TAG](#) and [CREATE EDGE](#).

```
# Create a tag and set the TTL options.
nebula> CREATE TAG t2(a int, b int, c string) ttl_duration= 100, ttl_col = "a";

# Insert a vertex with tag t2. The timeout timestamp is 1612778164774 (1612778164674 + 100).
nebula> INSERT VERTEX t2(a, b, c) values "102":(1612778164674, 30, "Hello");
```

Remove a timeout

To disable TTL and remove the timeout on a property, you can use the following approaches.

- Drop the property with the timeout.

```
nebula> ALTER TAG t1 DROP (a);
```

- Set `ttl_col` to an empty string.

```
nebula> ALTER TAG t1 ttl_col = "";
```

- Set `ttl_duration` to `0`. This operation keeps the TTL options and prevents the property from expiring and the property schema from being modified.

```
nebula> ALTER TAG t1 ttl_duration = 0;
```

Last update: August 5, 2021

4.7.6 WHERE

The `WHERE` clause filters the output by conditions.

The `WHERE` clause usually works in the following queries:

- Native nGQL: such as `GO` and `LOOKUP`.
- OpenCypher syntax: such as `MATCH` and `WITH`.

OpenCypher compatibility

- Using patterns in `WHERE` is not supported (TODO: planning), for example `WHERE (v)-->(v2)`.
- [Filtering on edge rank](#) is a native nGQL feature. To retrieve the rank value in openCypher statements, use the `rank()` function, such as `MATCH (:player)-[e:follow]->() RETURN rank(e);`.

Basic usage

Note

In the following examples, `$$` and `$$^` are reference operators. For more information, see [Operators](#).

DEFINE CONDITIONS WITH BOOLEAN OPERATORS

Use the boolean operators `NOT`, `AND`, `OR`, and `XOR` to define conditions in `WHERE` clauses. For the precedence of the operators, see [Precedence](#).

```
nebula> MATCH (v:player) \
  WHERE v.name == "Tim Duncan" \
  XOR (v.age < 30 AND v.name == "Yao Ming") \
  OR NOT (v.name == "Yao Ming" OR v.name == "Tim Duncan") \
  RETURN v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Marco Belinelli" | 32 |
| "Aron Baynes" | 32 |
| "LeBron James" | 34 |
| "James Harden" | 29 |
| "Manu Ginobili" | 41 |
+-----+-----+
...
```

```
nebula> GO FROM "player100" \
  OVER follow \
  WHERE follow.degree > 90 \
  OR $$.player.age != 33 \
  AND $$.player.name != "Tony Parker";
+-----+
| follow._dst |
+-----+
| "player101" |
| "player125" |
+-----+
```

FILTER ON PROPERTIES

Use vertex or edge properties to define conditions in `WHERE` clauses.

- Filter on a vertex property:

```
nebula> MATCH (v:player)-[e]->(v2) \
  WHERE v2.age < 25 \
  RETURN v2.name, v2.age;
+-----+-----+
| v2.name | v2.age |
+-----+-----+
| "Luka Doncic" | 20 |
| "Kristaps Porzingis" | 23 |
| "Ben Simmons" | 22 |
+-----+-----+
```

```
nebula> GO FROM "player100" \
  OVER follow \
  WHERE $^.player.age >= 42;
+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
| "player125" |
+-----+
```

- Filter on an edge property:

```
nebula> MATCH (v:player)-[e]->() \
  WHERE e.start_year < 2000 \
  RETURN DISTINCT v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Shaquille O'Neal" | 47 |
| "Steve Nash" | 45 |
| "Ray Allen" | 43 |
| "Grant Hill" | 46 |
| "Tony Parker" | 36 |
+-----+-----+
...
```

```
nebula> GO FROM "player100" \
  OVER follow \
  WHERE follow.degree > 90;
+-----+
| follow._dst |
+-----+
| "player101" |
+-----+
| "player125" |
+-----+
```

FILTER ON DYNAMICALLY-CALCULATED PROPERTIES

```
nebula> MATCH (v:player) \
  WHERE v[toLowerCase("AGE")] < 21 \
  RETURN v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Luka Doncic" | 20 |
+-----+-----+
```

FILTER ON EXISTING PROPERTIES

```
nebula> MATCH (v:player) \
  WHERE exists(v.age) \
  RETURN v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Boris Diaw" | 36 |
| "DeAndre Jordan" | 30 |
+-----+-----+
```

FILTER ON EDGE RANK

In nGQL, if a group of edges has the same source vertex, destination vertex, and properties, the only thing that distinguishes them is the rank. Use rank conditions in `WHERE` clauses to filter such edges.

```
# The following example creates test data.
nebula> CREATE SPACE test (vid_type=FIXED_STRING(30));
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);

# The following example use rank to filter edges and retrieves edges with a rank greater than 2.
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   |
-----
```

Filter on strings

Use `STARTS WITH`, `ENDS WITH`, or `CONTAINS` in `WHERE` clauses to match a specific part of a string. String matching is case-sensitive.

STARTS WITH

`STARTS WITH` will match the beginning of a string.

The following example uses `STARTS WITH "T"` to retrieve the information of players whose name starts with `T`.

```
nebula> MATCH (v:player) \
    WHERE v.name STARTS WITH "T" \
    RETURN v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Tracy McGrady" | 39 |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
| "Tim Duncan" | 42 |
+-----+-----+
| "Tiago Splitter" | 34 |
+-----+-----+
```

If you use `STARTS WITH "t"` in the preceding statement, an empty set is returned because no name in the dataset starts with the lowercase `t`.

```
nebula> MATCH (v:player) \
    WHERE v.name STARTS WITH "t" \
    RETURN v.name, v.age;
Empty set (time spent 5080/6474 us)
```

ENDS WITH

`ENDS WITH` will match the ending of a string.

The following example uses `ENDS WITH "r"` to retrieve the information of players whose name ends with `r`.

```
nebula> MATCH (v:player) \
    WHERE v.name ENDS WITH "r" \
    RETURN v.name, v.age;
+-----+-----+
```

```
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Vince Carter" | 42 |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
| "Tiago Splitter" | 34 |
+-----+-----+
```

CONTAINS

`CONTAINS` will match a certain part of a string.

The following example uses `CONTAINS "Pa"` to match the information of players whose name contains `Pa`.

```
nebula> MATCH (v:player) \
  WHERE v.name CONTAINS "Pa" \
  RETURN v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Paul George" | 28 |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
| "Paul Gasol" | 38 |
+-----+-----+
| "Chris Paul" | 33 |
+-----+-----+
```

NEGATIVE STRING MATCHING

You can use the boolean operator `NOT` to negate a string matching condition.

```
nebula> MATCH (v:player) \
  WHERE NOT v.name ENDS WITH "R" \
  RETURN v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Rajon Rondo" | 33 |
+-----+-----+
| "Rudy Gay" | 32 |
+-----+-----+
| "Dejounte Murray" | 29 |
+-----+-----+
| "Chris Paul" | 33 |
+-----+-----+
| "Carmelo Anthony" | 34 |
+-----+-----+
...
```

Filter on lists

MATCH VALUES IN A LIST

Use the `IN` operator to check if a value is in a specific list.

```
nebula> MATCH (v:player) \
  WHERE v.age IN range(20,25) \
  RETURN v.name, v.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Ben Simmons" | 22 |
+-----+-----+
| "Kristaps Porzingis" | 23 |
+-----+-----+
| "Luka Doncic" | 20 |
+-----+-----+
| "Kyle Anderson" | 25 |
+-----+-----+
| "Giannis Antetokounmpo" | 24 |
+-----+-----+
| "Joel Embiid" | 25 |
+-----+-----+

nebula> LOOKUP ON player WHERE player.age IN [25,28] YIELD player.name, player.age;
+-----+-----+
| VertexID | player.name | player.age |
+-----+-----+
| "player135" | "Damian Lillard" | 28 |
+-----+-----+
| "player131" | "Paul George" | 28 |
+-----+-----+
```

```
+-----+-----+-----+
| "player130" | "Joel Embiid" | 25   |
+-----+-----+-----+
| "player123" | "Ricky Rubio" | 28   |
+-----+-----+-----+
| "player106" | "Kyle Anderson" | 25   |
+-----+-----+-----+
```

MATCH VALUES NOT IN A LIST

Use `NOT` before `IN` to rule out the values in a list.

```
nebula> MATCH (v:player) \
    WHERE v.age NOT IN range(20,25) \
    RETURN v.name AS Name, v.age AS Age \
    ORDER BY Age;
+-----+-----+
| Name      | Age |
+-----+-----+
| "Kyrie Irving" | 26 |
+-----+-----+
| "Cory Joseph" | 27 |
+-----+-----+
| "Damian Lillard" | 28 |
+-----+-----+
| "Paul George" | 28 |
+-----+-----+
| "Ricky Rubio" | 28 |
+-----+-----+
...
```

.....

Last update: August 5, 2021

4.7.7 YIELD

`YIELD` defines the output of an nGQL query.

`YIELD` can lead a clause or a statement:

- A `YIELD` clause works in nGQL statements such as `GO`, `FETCH`, or `LOOKUP`.
- A `YIELD` statement works in a composite query or independently.

OpenCypher compatibility

This topic applies to native nGQL only. For the openCypher syntax, use `RETURN`.

`YIELD` has different functions in openCypher and nGQL.

- In openCypher, `YIELD` is used in the `CALL[...YIELD]` clause to specify the output of the procedure call.

Note

NGQL does not support `CALL[...YIELD]` yet.

- In nGQL, `YIELD` works like `RETURN` in openCypher.

Note

In the following examples, `$$` and `$-` are reference operators. For more information, see [Operators](#).

YIELD clauses

SYNTAX

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...];
```

Parameter	Description
<code>DISTINCT</code>	Aggregates the output and makes the statement return a distinct result set.
<code>col</code>	A field to be returned. If no alias is set, <code>col</code> will be a column name in the output.
<code>alias</code>	An alias for <code>col</code> . It is set after the keyword <code>AS</code> and will be a column name in the output.

USE A YIELD CLAUSE IN A STATEMENT

- Use `YIELD` with `GO`:

```
nebula> GO FROM "player100" OVER follow \
    YIELD $$.player.name AS Friend, $$.player.age AS Age;
+-----+-----+
| Friend | Age |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
```

```
| "Manu Ginobili" | 41 |
+-----+-----+
```

- Use `YIELD` with `FETCH`:

```
nebula> FETCH PROP ON player "player100" \
    YIELD player.name;
+-----+-----+
| VertexID | player.name |
+-----+-----+
| "player100" | "Tim Duncan" |
+-----+-----+
```

- Use `YIELD` with `LOOKUP`:

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

YIELD statements

SYNTAX

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...]
[WHERE <conditions>];
```

Parameter	Description
<code>DISTINCT</code>	Aggregates the output and makes the statement return a distinct result set.
<code>col</code>	A field to be returned. If no alias is set, <code>col</code> will be a column name in the output.
<code>alias</code>	An alias for <code>col</code> . It is set after the keyword <code>AS</code> and will be a column name in the output.
<code>conditions</code>	Conditions set in a <code>WHERE</code> clause to filter the output. For more information, see WHERE .

USE A YIELD STATEMENT IN A COMPOSITE QUERY

In a [composite query](#), a `YIELD` statement accepts, filters, and modifies the result set of the preceding statement, and then outputs it.

The following query finds the players that "player100" follows and calculates their average age.

```
nebula> GO FROM "player100" OVER follow \
    YIELD follow._dst AS ID \
    | FETCH PROP ON player $-.ID \
    YIELD player.age AS Age \
    | YIELD AVG($-.Age) as Avg_age, count(*)as Num_friends;
+-----+-----+
| Avg_age | Num_friends |
+-----+-----+
| 38.5    | 2          |
+-----+-----+
```

The following query finds the players that "player101" follows with the follow degrees greater than 90.

```
nebula> $var1 = GO FROM "player101" OVER follow \
    YIELD follow.degree AS Degree, follow._dst as ID; \
    YIELD $var1.ID AS ID WHERE $var1.Degree > 90;
+-----+
| ID   |
+-----+
| "player100" |
+-----+
| "player125" |
+-----+
```

USE A STANDALONE YIELD STATEMENT

A `YIELD` statement can calculate a valid expression and output the result.

```
nebula> YIELD rand32(1, 6);
+-----+
| rand32(1,6) |
+-----+
| 3           |
+-----+  
  
nebula> YIELD "He1" + "\tlo" AS string1, ", World!" AS string2;
+-----+-----+
| string1    | string2    |
+-----+-----+
| "He1      lo" | ", World!" |
+-----+-----+  
  
nebula> YIELD hash("Tim") % 100;
+-----+
| (hash(Tim)%100) |
+-----+
| 42           |
+-----+  
  
nebula> YIELD \
  CASE 2+3 \
  WHEN 4 THEN 0 \
  WHEN 5 THEN 1 \
  ELSE -1 \
  END \
  AS result;
+-----+
| result |
+-----+
| 1       |
+-----+
```

Last update: August 6, 2021

4.7.8 WITH

The `WITH` clause can retrieve the output from a query part, process it, and pass it to the next query part as the input.

OpenCypher compatibility

This topic applies to openCypher syntax only.

Note

`WITH` has a similar function with the [Pipe](#) symbol in native nGQL, but they work in different ways. DO NOT use pipe symbols in the openCypher syntax or use `WITH` in native nGQL statements.

Combine statements and form a composite query

Use a `WITH` clause to combine statements and transfer the output of a statement as the input of another statement.

EXAMPLE 1

The following statement:

1. Matches a path.
2. Outputs all the vertices on the path to a list with the `nodes()` function.
3. Unwinds the list into rows.
4. Removes duplicated vertices and returns a set of distinct vertices.

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})--() \
  WITH nodes(p) AS n \
  UNWIND n AS n1 \
  RETURN DISTINCT n1;
```

EXAMPLE 2

The following statement:

1. Matches the vertex with the VID `player100`.
2. Outputs all the tags of the vertex into a list with the `labels()` function.
3. Unwinds the list into rows.
4. Returns the output.

```
nebula> MATCH (v) \
  WHERE id(v)=="player100" \
  WITH labels(v) AS tags_unf \
  UNWIND tags_unf AS tags_f \
  RETURN tags_f;
```

tags_f
"star"
"player"
"person"

Filter composite queries

`WITH` can work as a filter in the middle of a composite query.

```
nebula> MATCH (v:player)-->(v2:player) \
  WITH DISTINCT v2 AS v2, v2.age AS Age \
  ORDER BY Age \
  WHERE Age<25 \
  RETURN v2.name AS Name, Age;
```

Name	Age
"Luka Doncic"	20
"Ben Simmons"	22
"Kristaps Porzingis"	23

Process the output before using `collect()`

Use a `WITH` clause to sort and limit the output before using `collect()` to transform the output into a list.

```
nebula> MATCH (v:player) \
  WITH v.name AS Name \
  ORDER BY Name DESC \
  LIMIT 3 \
  RETURN collect(Name);
```

collect(Name)
["Yao Ming", "Vince Carter", "Tracy McGrady"]

Use with `RETURN`

Set an alias using a `WITH` clause, and then output the result through a `RETURN` clause.

```
nebula> WITH [1, 2, 3] AS list  RETURN 3 IN list AS r;
+-----+
| r   |
+-----+
| true |
+-----+
```

```
nebula> WITH 4 AS one, 3 AS two RETURN one > two AS result;
+-----+
| result |
+-----+
| true   |
+-----+
```

Last update: August 6, 2021

4.8 Space statements

4.8.1 CREATE SPACE

Graph spaces are used to store data in a physically isolated way in Nebula Graph, which is similar to the database concept in MySQL. The `CREATE SPACE` statement creates a new graph space with the given name.

Prerequisites

Only the God role can use the `CREATE SPACE` statement. For more information, see [AUTHENTICATION](#).

Syntax

```
CREATE SPACE [IF NOT EXISTS] <graph_space_name> (
  [partition_num = <partition_number>],
  [replica_factor = <replica_number>],
  vid_type = {FIXED_STRING(<N>) | INT[64]}
)
[COMMENT = '<comment>'];
```

Parameter	Description
<code>IF NOT EXISTS</code>	Detects if the related graph space exists. If it does not exist, a new one will be created. The graph space existence detection here only compares the graph space name (excluding properties).
<code><graph_space_name></code>	Uniquely identifies a graph space in a Nebula Graph instance. The name of the graph space is case-sensitive and allows letters, numbers, or underscores. Keywords and reserved words are not allowed.
<code>partition_num</code>	Specifies the number of partitions in each replica. The suggested number is five times the number of the hard disks in the cluster. For example, if you have 3 hard disks in the cluster, we recommend that you set 15 partitions. The default value is 100.
<code>replica_factor</code>	Specifies the number of replicas in the cluster. The suggested number is 3 in a production environment and 1 in a test environment. The replica number must be an odd number for the need of quorum-based voting. The default value is 1.
<code>vid_type</code>	A required parameter. Specifies the VID type in a graph space. Available values are <code>FIXED_STRING(N)</code> and <code>INT64</code> . <code>INT</code> equals to <code>INT64</code> . <code>FIXED_STRING(<N>)</code> specifies the VID as a string, while <code>INT64</code> specifies it as an integer. <code>N</code> represents the maximum length of the VIDs. If you set a VID that is longer than <code>N</code> characters, Nebula Graph throws an error.
<code>COMMENT</code>	The remarks of the graph space. The maximum length is 256 bytes. By default, there is no comments on a space.

⚠ Caution

If the replica number is set to one, you will not be able to load balance or scale out the Nebula Graph Storage Service with the [BALANCE](#) statement.

⚠ Caution

Restrictions on VID type change and VID length

1. In Nebula Graph 1.x, the VID type can only be `INT64` and does not support string. In Nebula Graph 2.x, the VID type can be both `INT64` and `FIXED_STRING(<N>)`. You should specify the VID type when creating a graph space and keep consistency when using the `INSERT` statement. Otherwise, Nebula Graph throws `Wrong vertex id type: 1001`.
2. The length of the VID should not be longer than `N` characters. If it exceeds `N`, Nebula Graph throws `The VID must be a 64-bit integer or a string fitting space vertex id length limit..`

Legacy version compatibility

In the 2.x releases before 2.5.0, `vid_type` is not a required parameter and its default value is `FIXED_STRING(8)`.

Note

`graph_space_name`, `partition_num`, `replica_factor`, `vid_type`, and `comment` cannot be modified once set. To modify them, drop the current working graph space with `DROP SPACE` and create a new one with `CREATE SPACE`.

Examples

```
# The following example creates a graph space with a specified VID type and the maximum length. Other fields still use the default values.
nebula> CREATE SPACE my_space_1 (vid_type=FIXED_STRING(30));

# The following example creates a graph space with a specified partition number, replica number, and VID type.
nebula> CREATE SPACE my_space_2 (partition_num=15, replica_factor=1, vid_type=FIXED_STRING(30));

# The following example creates a graph space with a specified partition number, replica number, and VID type, and adds a comment on it.
nebula> CREATE SPACE my_space_3 (partition_num=15, replica_factor=1, vid_type=FIXED_STRING(30)) comment="图空间";
```

Implementation of the operation

Caution

Trying to use a newly created graph space may fail because the creation is implemented asynchronously.

Nebula Graph implements the creation in the next heartbeat cycle. To make sure the creation is successful, take one of the following approaches:

- Find the new graph space in the result of `SHOW SPACES` or `DESCRIBE SPACE`. If you cannot, wait a few seconds and try again.
- Wait for two heartbeat cycles, i.e., 20 seconds.

To change the heartbeat interval, modify the `heartbeat_interval_secs` parameter in the `configuration files` for all services. If the heartbeat interval is too short (i.e., less than 5 seconds), disconnection between peers may happen because of the misjudgment of machines in the distributed system.

Check partition distribution

On some large clusters, the partition distribution is possibly unbalanced because of the different startup times. You can run the following command to do a check of the machine distribution.

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 8 | "basketballplayer:3, test:5" | "basketballplayer:10, test:10" |
+-----+-----+-----+-----+-----+
| "storaged1" | 9779 | "ONLINE" | 9 | "basketballplayer:4, test:5" | "basketballplayer:10, test:10" |
+-----+-----+-----+-----+-----+
| "storaged2" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:10, test:10" |
+-----+-----+-----+-----+-----+
| "Total" | | 20 | "basketballplayer:10, test:10" | "basketballplayer:30, test:30" |
+-----+-----+-----+-----+-----+
```

To balance the request loads, use the following command.

```
nebula> BALANCE LEADER;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 7 | "basketballplayer:3, test:4" | "basketballplayer:10, test:10" |
+-----+-----+-----+-----+-----+
| "storaged1" | 9779 | "ONLINE" | 7 | "basketballplayer:4, test:3" | "basketballplayer:10, test:10" |
+-----+-----+-----+-----+-----+
| "storaged2" | 9779 | "ONLINE" | 6 | "basketballplayer:3, test:3" | "basketballplayer:10, test:10" |
+-----+-----+-----+-----+-----+
```

```
| "Total"      |           | 20           | "basketballplayer:10, test:10" | "basketballplayer:30, test:30" |
```

Last update: August 13, 2021

4.8.2 USE

`USE` specifies a graph space as the current working graph space for subsequent queries.

Prerequisites

Running the `USE` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.

Syntax

```
USE <graph_space_name>;
```

Examples

```
# The following example specifies space1 as the current working graph space.
nebula> USE space1;

# The following example traverses in space1.
nebula> GO FROM 1 OVER edge1;

# The following example specifies space2 as the current working graph space.
nebula> USE space2;

# The following example traverses in space2. Hereafter, you cannot read any data from space1, because these vertices and edges being traversed have no
relevance with space1.
nebula> GO FROM 2 OVER edge2;
```

⚠ Caution

You cannot use two graph spaces in one statement.

Different from Fabric Cypher, graph spaces in Nebula Graph are fully isolated from each other. Making a graph space as the working graph space prevents you from accessing other spaces. The only way to traverse in a new graph space is to switch by the `USE` statement. In Fabric Cypher, you can use two graph spaces in one statement (using the `USE + CALL` syntax). But in Nebula Graph, you can only use one graph space in one statement.

Last update: August 13, 2021

4.8.3 SHOW SPACES

`SHOW SPACES` lists all the graph spaces in the Nebula Graph examples.

Syntax

```
SHOW SPACES;
```

Example

```
nebula> SHOW SPACES;
+-----+
| Name      |
+-----+
| "cba"      |
+-----+
| "basketballplayer" |
+-----+
```

To create graph spaces, see [CREATE SPACE](#).

Last update: August 13, 2021

4.8.4 DESCRIBE SPACE

`DESCRIBE SPACE` returns the information about the specified graph space.

Syntax

You can use `DESC` instead of `DESCRIBE` for short.

```
DESC[RIBE] SPACE <graph_space_name>;
```

The `DESCRIBE SPACE` statement is different from the `SHOW SPACES` statement. For details about `SHOW SPACES`, see [SHOW SPACES](#).

Example

```
nebula> DESCRIBE SPACE basketballplayer;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | Name          | Partition Number | Replica Factor | Charset | Collate | Vid Type | Atomic Edge | Group | Comment |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | "basketballplayer" | 10            | 1              | "utf8"  | "utf8_bin" | "FIXED_STRING(32)" | false  | "default" |      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Last update: August 13, 2021

4.8.5 DROP SPACE

`DROP SPACE` deletes everything in the specified graph space.

Prerequisites

Only the God role can use the `DROP SPACE` statement. For more information, see [AUTHENTICATION](#).

Syntax

```
DROP SPACE [IF EXISTS] <graph_space_name>;
```

You can use the `IF EXISTS` keywords when dropping spaces. These keywords automatically detect if the related graph space exists. If it exists, it will be deleted. Otherwise, no graph space will be deleted.

The `DROP SPACE` statement does not immediately remove all the files and directories from the disk. You can specify another graph space with the `USE` statement and `submit job compact`.

Caution

BE CAUTIOUS about running the `DROP SPACE` statement.

Last update: August 13, 2021

4.9 Tag statements

4.9.1 CREATE TAG

`CREATE TAG` creates a tag with the given name in a graph space.

OpenCypher compatibility

Tags in nGQL are similar to labels in openCypher. But they are also quite different. For example, the ways to create them are different.

- In openCypher, labels are created together with vertices in `CREATE` statements.
- In nGQL, tags are created separately using `CREATE TAG` statements. Tags in nGQL are more like tables in MySQL.

Prerequisites

Running the `CREATE TAG` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.

Syntax

To create a tag in a specific graph space, you must specify the current working space with the `USE` statement.

```
CREATE TAG [IF NOT EXISTS] <tag_name>
(
  <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
  [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
)
[TTL_DURATION = <ttl_duration>]
[TTL_COL = <prop_name>]
[COMMENT = '<comment>'];
```

Parameter	Description
<code>IF NOT EXISTS</code>	Detects if the tag that you want to create exists. If it does not exist, a new one will be created. The tag existence detection here only compares the tag names (excluding properties).
<code><tag_name></code>	The tag name must be unique in a graph space. Once the tag name is set, it can not be altered. The rules for permitted tag names are the same as those for graph space names. For prohibited names, see Keywords and reserved words .
<code><prop_name></code>	The name of the property. It must be unique for each tag. The rules for permitted property names are the same as those for tag names.
<code><data_type></code>	Shows the data type of each property. For a full description of the property data types, see Data types and Boolean .
<code>NULL \ NOT NULL</code>	Specifies if the property supports <code>NULL NOT NULL</code> . The default value is <code>NULL</code> .
<code>DEFAULT</code>	Specifies a default value for a property. The default value can be a literal value or an expression supported by Nebula Graph. If no value is specified, the default value is used when inserting a new vertex.
<code>COMMENT</code>	The remarks of a certain property or the tag itself. The maximum length is 256 bytes. By default, there will be no comments on a tag.
<code>TTL_DURATION</code>	Specifies the life cycle for the property. The property that exceeds the specified TTL expires. The expiration threshold is the <code>TTL_COL</code> value plus the <code>TTL_DURATION</code> . The default value of <code>TTL_DURATION</code> is <code>0</code> . It means the data never expires.
<code>TTL_COL</code>	Specifies the property to set a timeout on. The data type of the property must be <code>int</code> or <code>timestamp</code> . A tag can only specify one field as <code>TTL_COL</code> . For more information on TTL, see TTL options .

EXAMPLES

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

# The following example creates a tag with no properties.
nebula> CREATE TAG no_property();

# The following example creates a tag with a default value.
nebula> CREATE TAG player_with_default(name string, age int DEFAULT 20);

# In the following example, the TTL of the create_time field is set to be 100 seconds.
nebula> CREATE TAG woman(name string, age int, \
    married bool, salary double, create_time timestamp) \
    TTL_DURATION = 100, TTL_COL = "create_time";
```

Implementation of the operation

Trying to use a newly created tag may fail because the creation of the tag is implemented asynchronously.

Nebula Graph implements the creation of the tag in the next heartbeat cycle. To make sure the creation is successful, take one of the following approaches:

- Find the new tag in the result of [SHOW TAGS](#). If you cannot, wait a few seconds and try again.
- Wait for two heartbeat cycles, i.e., 20 seconds.

To change the heartbeat interval, modify the `heartbeat_interval_secs` parameter in the [configuration files](#) for all services.

Last update: August 13, 2021

4.9.2 DROP TAG

`DROP TAG` drops a tag with the given name in the current working graph space.

A vertex can have one or more tags.

- If a vertex has only one tag, the vertex **CANNOT** be accessed after you drop it. But its edges are available. The vertex will be deleted in the next compaction.
- If a vertex has multiple tags, the vertex is still accessible after you drop one of them. But all the properties defined by this dropped tag **CANNOT** be accessed.

This operation only deletes the Schema data. All the files or directories in the disk will not be deleted directly until the next compaction.

Prerequisites

- Running the `DROP TAG` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.
- Before you drop a tag, make sure that the tag does not have any indexes. Otherwise, the conflict error (`[ERROR (-8)]: Conflict!`) will be returned when you run the `DROP TAG` statement. To drop an index, see [DROP INDEX](#).

Syntax

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

- `IF NOT EXISTS` : Detects if the tag that you want to drop exists. Only when it exists will it be dropped.
- `tag_name` : Specifies the tag name that you want to drop. You can drop only one tag in one statement.

Example

```
nebula> CREATE TAG test(p1 string, p2 int);
nebula> DROP TAG test;
```

Note

In nGQL, there is no such statement to drop a certain tag of a vertex with the given name.

- In openCypher, you can use the statement `REMOVE v:LABEL` to drop the tag `LABEL` of the vertex `v`.
- In nGQL, after `CREATE TAG` and `INSERT VERTEX`, you can add a `TAG` on the vertex. But there is no way to drop the `TAG` afterward.

We recommend you to add a field to identify the logical deletion in the schema. For example, add `removed` to the schema of each tag.

Last update: August 13, 2021

4.9.3 ALTER TAG

ALTER TAG alters the structure of a tag with the given name in a graph space. You can add or drop properties, and change the data type of an existing property. You can also set a [TTL](#) (Time-To-Live) on a property, or change its TTL duration.

Prerequisites

- Running the `ALTER TAG` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.
- Before you alter properties for a tag, make sure that the properties are not indexed. If the properties contain any indexes, the conflict error `[ERROR (-8)]: Conflict!` will occur when you `ALTER TAG`. For more information on dropping an index, see [DROP INDEX](#).

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
```

- `tag_name` : Specifies the tag name that you want to alter. You can alter only one tag in one statement. Before you alter a tag, make sure that the tag exists in the current working graph space. If the tag does not exist, an error will occur when you alter it.
- Multiple `ADD`, `DROP`, and `CHANGE` clauses are permitted in a single `ALTER TAG` statement, separated by commas.

Examples

```
nebula> CREATE TAG t1 (p1 string, p2 int);
nebula> ALTER TAG t1 ADD (p3 int, p4 string);
nebula> ALTER TAG t1 TTL_DURATION = 2, TTL_COL = "p2";
```

Implementation of the operation

Trying to use a newly altered tag may fail because the alteration of the tag is implemented asynchronously.

Nebula Graph implements the alteration of the tag in the next heartbeat cycle. To make sure the alteration is successful, take one of the following approaches:

- Use `DESCRIBE TAG` to confirm that the tag information is updated. If it is not, wait a few seconds and try again.
- Wait for two heartbeat cycles, i.e., 20 seconds.

To change the heartbeat interval, modify the `heartbeat_interval_secs` parameter in the [configuration files](#) for all services.

Last update: August 13, 2021

4.9.4 SHOW TAGS

The `SHOW TAGS` statement shows the name of all tags in the current graph space.

You do not need any privileges for the graph space to run the `SHOW TAGS` statement. But the returned results are different based on [role privileges](#).

Syntax

```
SHOW TAGS;
```

Examples

```
nebula> SHOW TAGS;
+-----+
| Name   |
+-----+
| "player" |
+-----+
| "team"   |
+-----+
```

Last update: August 13, 2021

4.9.5 DESCRIBE TAG

`DESCRIBE TAG` returns the information about a tag with the given name in a graph space, such as field names, data type, and so on.

Prerequisite

Running the `DESCRIBE TAG` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.

Syntax

```
DESC[RIBE] TAG <tag_name>;
```

You can use `DESC` instead of `DESCRIBE` for short.

Example

```
nebula> DESCRIBE TAG player;
+-----+-----+-----+-----+
| Field | Type   | Null | Default | Comment |
+-----+-----+-----+-----+
| "name" | "string" | "YES" |          |          |
+-----+-----+-----+-----+
| "age"  | "int64"  | "YES" |          |          |
+-----+-----+-----+-----+
```

Last update: August 13, 2021

4.10 Edge type statements

4.10.1 CREATE EDGE

`CREATE EDGE` creates an edge type with the given name in a graph space.

OpenCypher compatibility

Edge types in nGQL are similar to relationship types in openCypher. But they are also quite different. For example, the ways to create them are different.

- In openCypher, relationship types are created together with vertices in `CREATE` statements.
- In nGQL, edge types are created separately using `CREATE EDGE` statements. Edge types in nGQL are more like tables in MySQL.

Prerequisites

Running the `CREATE EDGE` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.

Syntax

To create an edge type in a specific graph space, you must specify the current working space with the `USE` statement.

```
CREATE EDGE [IF NOT EXISTS] <edge_type_name>
(
  <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
  [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
)
[TTL_DURATION = <ttl_duration>]
[TTL_COL = <prop_name>]
[COMMENT = '<comment>'];
```

Parameter	Description
<code>IF NOT EXISTS</code>	Detects if the edge type that you want to create exists. If it does not exist, a new one will be created. The edge type existence detection here only compares the edge type names (excluding properties).
<code><edge_type_name></code>	The edge type name must be unique in a graph space. Once the edge type name is set, it can not be altered. The rules for permitted edge type names are the same as those for graph space names. For prohibited names, see Keywords and reserved words .
<code><prop_name></code>	The name of the property. It must be unique for each edge type. The rules for permitted property names are the same as those for edge type names.
<code><data_type></code>	Shows the data type of each property. For a full description of the property data types, see Data types and Boolean .
<code>NULL \ NOT NULL</code>	Specifies if the property supports <code>NULL NOT NULL</code> . The default value is <code>NULL</code> .
<code>DEFAULT</code>	Specifies a default value for a property. The default value can be a literal value or an expression supported by Nebula Graph. If no value is specified, the default value is used when inserting a new edge.
<code>COMMENT</code>	The remarks of a certain property or the edge type itself. The maximum length is 256 bytes. By default, there will be no comments on an edge type.
<code>TTL_DURATION</code>	Specifies the life cycle for the property. The property that exceeds the specified TTL expires. The expiration threshold is the <code>TTL_COL</code> value plus the <code>TTL_DURATION</code> . The default value of <code>TTL_DURATION</code> is <code>0</code> . It means the data never expires.
<code>TTL_COL</code>	Specifies the property to set a timeout on. The data type of the property must be <code>int</code> or <code>timestamp</code> . An edge type can only specify one field as <code>TTL_COL</code> . For more information on TTL, see TTL options .

EXAMPLES

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

# The following example creates an edge type with no properties.
nebula> CREATE EDGE no_property();

# The following example creates an edge type with a default value.
nebula> CREATE EDGE follow_with_default(degree int DEFAULT 20);

# In the following example, the TTL of the p2 field is set to be 100 seconds.
nebula> CREATE EDGE e1(p1 string, p2 int, p3 timestamp) \
    TTL_DURATION = 100, TTL_COL = "p2";
```

Implementation of the operation

Trying to use a newly created edge type may fail because the creation of the edge type is implemented asynchronously.

Nebula Graph implements the creation of the edge type in the next heartbeat cycle. To make sure the creation is successful, take the following approaches:

- Find the new edge type in the result of [SHOW EDGES](#). If you cannot, wait a few seconds and try again.
- Wait for two heartbeat cycles, i.e., 20 seconds.

To change the heartbeat interval, modify the `heartbeat_interval_secs` parameter in the [configuration files](#) for all services.

Last update: August 13, 2021

4.10.2 DROP EDGE

`DROP EDGE` drops an edge type with the given name in a graph space.

An edge can have only one edge type. After you drop it, the edge **CANNOT** be accessed. The edge will be deleted in the next compaction.

This operation only deletes the Schema data. All the files or directories in the disk will not be deleted directly until the next compaction.

Prerequisites

- Running the `DROP EDGE` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.
- Before you drop an edge type, make sure that the edge type does not have any indexes. Otherwise, the conflict error (`[ERROR (-8)]: Conflict!`) will be returned. To drop an index, see [DROP INDEX](#).

Syntax

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

Edge type name

- `IF NOT EXISTS` : Detects if the edge type that you want to drop exists. Only when it exists will it be dropped.
- `edge_type_name` : Specifies the edge type name that you want to drop. You can drop only one edge type in one statement.

Example

```
nebula> CREATE EDGE e1(p1 string, p2 int);
nebula> DROP EDGE e1;
```

Last update: August 13, 2021

4.10.3 ALTER EDGE

`ALTER EDGE` alters the structure of an edge type with the given name in a graph space. You can add or drop properties, and change the data type of an existing property. You can also set a [TTL](#) (Time-To-Live) on a property, or change its TTL duration.

Prerequisites

- Running the `ALTER EDGE` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.
- Before you alter properties for an edge type, make sure that the properties are not indexed. If the properties contain any indexes, the conflict error `[ERROR (-8)]: Conflict!` will occur when you `ALTER EDGE`. For more information on dropping an index, see [DROP INDEX](#).

Syntax

```
ALTER EDGE <edge_type_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
```

- `edge_type_name` : Specifies the edge type name that you want to alter. You can alter only one edge type in one statement. Before you alter an edge type, make sure that the edge type exists in the graph space. If the edge type does not exist, an error occurs when you alter it.
- Multiple `ADD`, `DROP`, and `CHANGE` clauses are permitted in a single `ALTER EDGE` statement, separated by commas.

Example

```
nebula> CREATE EDGE e1(p1 string, p2 int);
nebula> ALTER EDGE e1 ADD (p3 int, p4 string);
nebula> ALTER EDGE e1 TTL_DURATION = 2, TTL_COL = "p2";
```

Implementation of the operation

Trying to use a newly altered edge type may fail because the alteration of the edge type is implemented asynchronously.

Nebula Graph implements the alteration of the edge type in the next heartbeat cycle. To make sure the alteration is successful, take one of the following approaches:

- Use [DESCRIBE EDGE](#) to confirm that the edge type information is updated. If it is not, wait a few seconds and try again.
- Wait for two heartbeat cycles, i.e., 20 seconds.

To change the heartbeat interval, modify the `heartbeat_interval_secs` parameter in the [configuration files](#) for all services.

Last update: August 13, 2021

4.10.4 SHOW EDGES

`SHOW EDGES` shows all edge types in the current graph space.

You do not need any privileges for the graph space to run the `SHOW EDGES` statement. But the returned results are different based on [role privileges](#).

Syntax

```
SHOW EDGES;
```

Example

```
nebula> SHOW EDGES;
+-----+
| Name   |
+-----+
| "follow" |
+-----+
| "serve"  |
+-----+
```

Last update: August 13, 2021

4.10.5 DESCRIBE EDGE

`DESCRIBE EDGE` returns the information about an edge type with the given name in a graph space, such as field names, data type, and so on.

Prerequisites

Running the `DESCRIBE EDGE` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.

Syntax

```
DESC[RIBE] EDGE <edge_type_name>
```

You can use `DESC` instead of `DESCRIBE` for short.

Example

```
nebula> DESCRIBE EDGE follow;
+-----+-----+-----+-----+
| Field | Type   | Null  | Default | Comment |
+-----+-----+-----+-----+
| "degree" | "int64" | "YES" |          |
+-----+-----+-----+-----+
```

Last update: August 13, 2021

4.11 Vertex statements

4.11.1 INSERT VERTEX

The `INSERT VERTEX` statement inserts one or more vertices into a graph space in Nebula Graph.

Prerequisites

Running the `INSERT VERTEX` statement requires some [privileges](#) for the graph space. Otherwise, Nebula Graph throws an error.

Syntax

```
INSERT VERTEX [IF NOT EXISTS] <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] ...]
```

- `IF NOT EXISTS` detects if the VID that you want to insert exists. If it does not exist, a new one will be inserted.

Note

- `IF NOT EXISTS` only compares the names of the VID and the tag (excluding properties). - `IF NOT EXISTS` will read to check whether the data exists, which will have a significant impact on performance.

- `tag_name` denotes the tag (vertex type), which must be created before `INSERT VERTEX`. For more information, see [CREATE TAG](#).
- `prop_name_list` contains the names of the properties on the tag.
- `VID` is the vertex ID. In Nebula Graph 2.0, string and integer VID types are supported. The VID type is set when a graph space is created. For more information, see [CREATE SPACE](#).
- `prop_value_list` must provide the property values according to the `prop_name_list`. If the property values do not match the data type in the tag, an error is returned. When the `NOT NULL` constraint is set for a given property, an error is returned if no property is given. When the default value for a property is `NULL`, you can omit to specify the property value. For details, see [CREATE TAG](#).

Caution

`INSERT VERTEX` and `CREATE` have different semantics.

- The semantics of `INSERT VERTEX` is closer to that of `INSERT` in NoSQL (key-value), or `UPSERT (UPDATE OR INSERT)` in SQL.
- When two `INSERT` statements (neither uses `IF NOT EXISTS`) with the same `VID` and `TAG` are operated at the same time, the latter `INSERT` will overwrite the former.
- When two `INSERT` statements with the same `VID` but different `TAGS` are operated at the same time, the operation of different tags will not overwrite each other.

Examples are as follows.

Examples

```
# The following examples create tag t1 with no property and inserts vertex "10" with no property.
nebula> CREATE TAG t1();
nebula> INSERT VERTEX t1() VALUE "10":();

nebula> CREATE TAG t2 (name string, age int);
nebula> INSERT VERTEX t2 (name, age) VALUES "11":("n1", 12);

# In the following example, the insertion fails because "a13" is not int.
nebula> INSERT VERTEX t2 (name, age) VALUES "12":("n1", "a13");

# The following example inserts two vertices at one time.
nebula> INSERT VERTEX t2 (name, age) VALUES "13":("n3", 12), "14":("n4", 8);

nebula> CREATE TAG t3(p1 int);
nebula> CREATE TAG t4(p2 string);

# The following example inserts vertex "21" with two tags.
nebula> INSERT VERTEX t3 (p1), t4(p2) VALUES "21": (321, "hello");
```

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

```
# The following examples insert vertex "11" with new values for multiple times.
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);
nebula> FETCH PROP ON t2 "11";
+-----+
| vertices_
|-----+
| ("11" :t2{age: 15, name: "n4"}) |
+-----+

nebula> CREATE TAG t5(p1 fixed_string(5) NOT NULL, p2 int, p3 int DEFAULT NULL);
nebula> INSERT VERTEX t5(p1, p2, p3) VALUES "001":("Abe", 2, 3);

# In the following example, the insertion fails because the value of p1 cannot be NULL.
nebula> INSERT VERTEX t5(p1, p2, p3) VALUES "002":(NULL, 4, 5);
[ERROR (-8)]: Storage Error: The not null field cannot be null.

# In the following example, the value of p3 is the default NULL.
nebula> INSERT VERTEX t5(p1, p2) VALUES "003":("cd", 5);
nebula> FETCH PROP ON t5 "003";
+-----+
| vertices_
|-----+
| ("003" :t5{p1: "cd", p2: 5, p3: __NULL__}) |
+-----+

# In the following example, the allowed maximum length of p1 is 5.
nebula> INSERT VERTEX t5(p1, p2) VALUES "004":("shalalalala", 4);
nebula> FETCH PROP ON t5 "004";
+-----+
| vertices_
|-----+
| ("004" :t5{p1: "shalalalala", p2: 4, p3: __NULL__}) |
+-----+
```

If you insert a vertex that already exists with `IF NOT EXISTS`, there will be no modification.

```
# The following example inserts vertex "1".
nebula> INSERT VERTEX t2 (name, age) VALUES "1":("n2", 13);
# Modify vertex "1" with IF NOT EXISTS. But there will be no modification as vertex "1" already exists.
nebula> INSERT VERTEX IF NOT EXISTS t2 (name, age) VALUES "1":("n3", 14);
nebula> FETCH PROP ON t2 "1";
+-----+
| vertices_
|-----+
| ("1" :t2{age: 13, name: "n2"}) |
+-----+
```

Last update: August 17, 2021

4.11.2 DELETE VERTEX

The `DELETE VERTEX` statement deletes vertices and the related incoming and outgoing edges of the vertices.

The `DELETE VERTEX` statement deletes one vertex or multiple vertices at a time. You can use `DELETE VERTEX` together with pipes. For more information about pipe, see [Pipe operator](#).

Syntax

```
DELETE VERTEX <vid> [, <vid> ...];
```

Examples

This query deletes the vertex whose ID is "team1".

```
nebula> DELETE VERTEX "team1";
```

This query shows that you can use `DELETE VERTEX` together with pipe to delete vertices.

```
nebula> GO FROM "player100" OVER serve WHERE serve.start_year == "2021" YIELD serve._dst AS id | DELETE VERTEX $-.id;
```

Delete the process and the related edges

Nebula Graph traverses the incoming and outgoing edges related to the vertices and deletes them all. Then Nebula Graph deletes the vertices.

Caution

- Atomic deletion is not supported during the entire process for now. Please retry when a failure occurs to avoid partial deletion, which will cause pendent edges.
- Deleting a supernode takes a lot of time. To avoid connection timeout before the deletion is complete, you can modify the parameter `--storage_client_timeout_ms` in `nebula-graphd.conf` to extend the timeout period.

Last update: August 17, 2021

4.11.3 UPDATE VERTEX

The `UPDATE VERTEX` statement updates properties on tags of a vertex.

In Nebula Graph, `UPDATE VERTEX` supports compare-and-set (CAS).

Note

An `UPDATE VERTEX` statement can only update properties on **ONE TAG** of a vertex.

Syntax

```
UPDATE VERTEX ON <tag_name> <vid>
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

Parameter	Required	Description	Example
<code>ON <tag_name></code>	Yes	Specifies the tag of the vertex. The properties to be updated must be on this tag.	<code>ON player</code>
<code><vid></code>	Yes	Specifies the ID of the vertex to be updated.	<code>"player100"</code>
<code>SET</code> <code><update_prop></code>	Yes	Specifies the properties to be updated and how they will be updated.	<code>SET age = age +1</code>
<code>WHEN</code> <code><condition></code>	No	Specifies the filter conditions. If <code><condition></code> evaluates to <code>false</code> , the <code>SET</code> clause will not take effect.	<code>WHEN name == "Tim"</code>
<code>YIELD</code> <code><output></code>	No	Specifies the output format of the statement.	<code>YIELD name AS Name</code>

Example

```
// This query checks the properties of vertex "player101".
nebula> FETCH PROP ON player "player101";
+-----+
| vertices_
+-----+
| ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----+


// This query updates the age property and returns name and the new age.
nebula> UPDATE VERTEX ON player "player101" \
    SET age = age + 2 \
    WHEN name == "Tony Parker" \
    YIELD name AS Name, age AS Age;
+-----+-----+
| Name      | Age   |
+-----+-----+
| "Tony Parker" | 38   |
+-----+-----+
```

Last update: August 17, 2021

4.11.4 UPSERT VERTEX

The `UPSERT` statement is a combination of `UPDATE` and `INSERT`. You can use `UPSERT VERTEX` to update the properties of a vertex if it exists or insert a new vertex if it does not exist.

Note

An `UPSERT VERTEX` statement can only update the properties on **ONE TAG** of a vertex.

The performance of `UPSERT` is much lower than that of `INSERT` because `UPSERT` is a read-modify-write serialization operation at the partition level.

Danger

Don't use `UPSERT` for scenarios with highly concurrent writes. You can use `UPDATE` or `INSERT` instead.

Syntax

```
UPSERT VERTEX ON <tag> <vid>
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

Parameter	Required	Description	Example
<code>ON <tag></code>	Yes	Specifies the tag of the vertex. The properties to be updated must be on this tag.	<code>ON player</code>
<code><vid></code>	Yes	Specifies the ID of the vertex to be updated or inserted.	<code>"player100"</code>
<code>SET <update_prop></code>	Yes	Specifies the properties to be updated and how they will be updated.	<code>SET age = age +1</code>
<code>WHEN <condition></code>	No	Specifies the filter conditions.	<code>WHEN name == "Tim"</code>
<code>YIELD <output></code>	No	Specifies the output format of the statement.	<code>YIELD name AS Name</code>

Insert a vertex if it does not exist

If a vertex does not exist, it is created no matter the conditions in the `WHEN` clause are met or not, and the `SET` clause always takes effect. The property values of the new vertex depend on:

- How the `SET` clause is defined.
- Whether the property has a default value.

For example, if:

- The vertex to be inserted will have properties `name` and `age` based on the tag `player`.
- The `SET` clause specifies that `age = 30`.

Then the property values in different cases are listed as follows:

Are <code>when</code> conditions met	If properties have default values	Value of <code>name</code>	Value of <code>age</code>
Yes	Yes	The default value	30
Yes	No	NULL	30
No	Yes	The default value	30
No	No	NULL	30

Here are some examples:

```
// This query checks if the following three vertices exist. The result "Empty set" indicates that the vertices do not exist.
nebula> FETCH PROP ON * "player666", "player667", "player668";
Empty set

nebula> UPSERT VERTEX ON player "player666" \
    SET age = 30 \
    WHEN name == "Joe" \
    YIELD name AS Name, age AS Age;
+-----+-----+
| Name | Age |
+-----+-----+
| __NULL__ | 30 |
+-----+-----+

nebula> UPSERT VERTEX ON player "player666" SET age = 31 WHEN name == "Joe" YIELD name AS Name, age AS Age;
+-----+-----+
| Name | Age |
+-----+-----+
| __NULL__ | 30 |
+-----+-----+

nebula> UPSERT VERTEX ON player "player667" \
    SET age = 31 \
    YIELD name AS Name, age AS Age;
+-----+-----+
| Name | Age |
+-----+-----+
| __NULL__ | 31 |
+-----+-----+

nebula> UPSERT VERTEX ON player "player668" \
    SET name = "Amber", age = age + 1 \
    YIELD name AS Name, age AS Age;
+-----+-----+
| Name | Age |
+-----+-----+
| "Amber" | __NULL__ |
+-----+-----+
```

In the last query of the preceding examples, since `age` has no default value, when the vertex is created, `age` is `NULL`, and `age = age + 1` does not take effect. But if `age` has a default value, `age = age + 1` will take effect. For example:

```
nebula> CREATE TAG player_with_default(name string, age int DEFAULT 20);
Execution succeeded

nebula> UPSERT VERTEX ON player_with_default "player101" \
    SET age = age + 1 \
    YIELD name AS Name, age AS Age;

+-----+-----+
| Name      | Age   |
+-----+-----+
| __NULL__ | 21   |
+-----+-----+
```

Update a vertex if it exists

If the vertex exists and the `WHEN` conditions are met, the vertex is updated.

```

WHEN name == "Tony Parker" \
YIELD name AS Name, age AS Age;
+-----+-----+
| Name      | Age |
+-----+-----+
| "Tony Parker" | 44 |
+-----+-----+

```

If the vertex exists and the `WHEN` conditions are not met, the update does not take effect.

```

nebula> FETCH PROP ON player "player101";
+-----+-----+
| vertices_          |
+-----+-----+
| ("player101" :player{age: 44, name: "Tony Parker"}) |
+-----+-----+
nebula> UPSERT VERTEX ON player "player101" \
    SET age = age + 2 \
    WHEN name == "Someone else" \
    YIELD name AS Name, age AS Age;
+-----+-----+
| Name      | Age |
+-----+-----+
| "Tony Parker" | 44 |
+-----+-----+

```

Last update: August 17, 2021

4.12 Edge statements

4.12.1 INSERT EDGE

The `INSERT EDGE` statement inserts an edge or multiple edges into a graph space from a source vertex (given by `src_vid`) to a destination vertex (given by `dst_vid`) with a specific rank in Nebula Graph.

When inserting an edge that already exists, `INSERT VERTEX` **overrides** the edge.

Syntax

```
INSERT EDGE [IF NOT EXISTS] <edge_type> ( <prop_name_list> ) {VALUES | VALUE}
<src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> )
[, <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ), ...];

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

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

- `IF NOT EXISTS` detects if the edge that you want to insert exists. If it does not exist, a new one will be inserted.

Note

- `IF NOT EXISTS` only detects whether exist and does not detect whether the property values overlap. - `IF NOT EXISTS` will read to check whether the data exists, which will have a significant impact on performance.

- `<edge_type>` denotes the edge type, which must be created before `INSERT EDGE`. Only one edge type can be specified in this statement.
- `<prop_name_list>` is the property name list in the given `<edge_type>`.
- `src_vid` is the VID of the source vertex. It specifies the start of an edge.
- `dst_vid` is the VID of the destination vertex. It specifies the end of an edge.
- `rank` is optional. It specifies the edge rank of the same edge type. If not specified, the default value is `0`. You can insert many edges with the same edge type, source vertex, and destination vertex by using different rank values.

OpenCypher compatibility

OpenCypher has no such concept as rank.

- `<prop_value_list>` must provide the value list according to `<prop_name_list>`. If the property values do not match the data type in the edge type, an error is returned. When the `NOT NULL` constraint is set for a given property, an error is returned if no property is given. When the default value for a property is `NULL`, you can omit to specify the property value. For details, see [CREATE EDGE](#).

Examples

```
# The following example creates edge type e1 with no property and inserts an edge from vertex "10" to vertex "11" with no property.
nebula> CREATE EDGE e1();
nebula> INSERT EDGE e1 () VALUES "10"->"11":();

# The following example inserts an edge from vertex "10" to vertex "11" with no property. The edge rank is 1.
nebula> INSERT EDGE e1 () VALUES "10"->"11":1:();

nebula> CREATE EDGE e2 (name string, age int);
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", 1);

# The following example creates edge type e2 with two properties.
```

```
nebula> INSERT EDGE e2 (name, age) VALUES \
  "12"->"13":("n1", 1), "13"->"14":("n2", 2);

# In the following example, the insertion fails because "a13" is not int.
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", "a13");
```

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

```
The following examples insert edge e2 with the new values for multiple times.
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);
nebula> FETCH PROP ON e2 "11"->"13";
+-----+
| edges_          |
+-----+
| [:e2 "11"->"13" @0 {age: 14, name: "n1"}] |
+-----+
```

If you insert an edge that already exists with `IF NOT EXISTS`, there will be no modification.

```
# The following example inserts edge e2 from vertex "14" to vertex "15".
nebula> INSERT EDGE e2 (name, age) VALUES "14"->"15">@1:("n1", 12);
# The following example alters the edge with IF NOT EXISTS. But there will be no alteration because edge e2 already exists.
nebula> INSERT EDGE IF NOT EXISTS e2 (name, age) VALUES "14"->"15">@1:("n2", 13);
nebula> FETCH PROP ON e2 "14"->"15">@1;
+-----+
| edges_          |
+-----+
| [:e2 "14"->"15" @1 {age: 12, name: "n1"}] |
+-----+
```

Note

- Nebula Graph 2.5.0 allows dangling edges. Therefore, you can write the edge before the source vertex or the destination vertex exists. At this time, you can get the (not written) vertex VID through `<edgetype>._src` or `<edgetype>._dst` (which is not recommended).
- Atomic operation is not guaranteed during the entire process for now. If it fails, please try again. Otherwise, partial writing will occur. At this time, the behavior of reading the data is undefined.
- Concurrently writing the same edge will cause an `edge conflict` error, so please try again later.
- The inserting speed of an edge is about half that of a vertex. Because in the storaged process, the insertion of an edge involves two tasks, while the insertion of a vertex involves only one task.

Last update: August 18, 2021

4.12.2 DELETE EDGE

The `DELETE EDGE` statement deletes one edge or multiple edges at a time. You can use `DELETE EDGE` together with pipe operators. For more information, see [PIPE OPERATORS](#).

To delete all the outgoing edges for a vertex, please delete the vertex. For more information, see [DELETE VERTEX](#).

Note

Atomic operation is not guaranteed during the entire process for now, so please retry when a failure occurs.

Syntax

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

Examples

```
nebula> DELETE EDGE serve "player100" -> "team204"@0;
```

The following example shows that you can use `DELETE EDGE` together with pipe operators to delete edges that meet the conditions.

```
nebula> GO FROM "player100" OVER follow \
WHERE follow._dst == "team204" \
YIELD follow._src AS src, follow._dst AS dst, follow._rank AS rank \
| DELETE EDGE follow $-src->$-dst @ $-rank;
```

Last update: August 17, 2021

4.12.3 UPDATE EDGE

The `UPDATE EDGE` statement updates properties on an edge.

In Nebula Graph, `UPDATE EDGE` supports compare-and-set (CAS).

Syntax

```
UPDATE EDGE ON <edge_type>
<src_vid> -> <dst_vid> [@<rank>]
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

Parameter	Required	Description	Example
ON <edge_type>	Yes	Specifies the edge type. The properties to be updated must be on this edge type.	ON serve
<src_vid>	Yes	Specifies the source vertex ID of the edge.	"player100"
<dst_vid>	Yes	Specifies the destination vertex ID of the edge.	"team204"
<rank>	No	Specifies the rank of the edge.	10
SET <update_prop>	Yes	Specifies the properties to be updated and how they will be updated.	SET start_year = start_year +1
WHEN <condition>	No	Specifies the filter conditions. If <condition> evaluates to <code>false</code> , the <code>SET</code> clause does not take effect.	WHEN end_year < 2010
YIELD <output>	No	Specifies the output format of the statement.	YIELD start_year AS Start_Year

Example

The following example checks the properties of the edge with the `GO` statement.

```
nebula> GO FROM "player100" \
    OVER serve \
    YIELD serve.start_year, serve.end_year;
+-----+-----+
| serve.start_year | serve.end_year |
+-----+-----+
| 1997           | 2016           |
+-----+-----+
```

The following example updates the `start_year` property and returns the `end_year` and the new `start_year`.

```
nebula> UPDATE EDGE on serve "player100" -> "team204" @0 \
    SET start_year = start_year + 1 \
    WHEN end_year > 2010 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| 1998       | 2016       |
+-----+-----+
```

Last update: August 17, 2021

4.12.4 UPSERT EDGE

The `UPSERT` statement is a combination of `UPDATE` and `INSERT`. You can use `UPSERT EDGE` to update the properties of an edge if it exists or insert a new edge if it does not exist.

The performance of `UPSERT` is much lower than that of `INSERT` because `UPSERT` is a read-modify-write serialization operation at the partition level.

Danger

Do not use `UPSERT` for scenarios with highly concurrent writes. You can use `UPDATE` or `INSERT` instead.

Syntax

```
UPSERT EDGE ON <edge_type>
<src_vid> -> <dst_vid> [<rank>]
SET <update_prop>
[WHEN <condition>]
[YIELD <properties>]
```

Parameter	Required	Description	Example
<code>ON <edge_type></code>	Yes	Specifies the edge type. The properties to be updated must be on this edge type.	<code>ON serve</code>
<code><src_vid></code>	Yes	Specifies the source vertex ID of the edge.	<code>"player100"</code>
<code><dst_vid></code>	Yes	Specifies the destination vertex ID of the edge.	<code>"team204"</code>
<code><rank></code>	No	Specifies the rank of the edge.	<code>10</code>
<code>SET <update_prop></code>	Yes	Specifies the properties to be updated and how they will be updated.	<code>SET start_year = start_year +1</code>
<code>WHEN <condition></code>	No	Specifies the filter conditions.	<code>WHEN end_year < 2010</code>
<code>YIELD <output></code>	No	Specifies the output format of the statement.	<code>YIELD start_year AS Start_Year</code>

Insert an edge if it does not exist

If an edge does not exist, it is created no matter the conditions in the `WHEN` clause are met or not, and the `SET` clause takes effect. The property values of the new edge depend on:

- How the `SET` clause is defined.
- Whether the property has a default value.

For example, if:

- The edge to be inserted will have properties `start_year` and `end_year` based on the edge type `serve`.
- The `SET` clause specifies that `end_year = 2021`.

Then the property values in different cases are listed as follows:

Are WHEN conditions met	If properties have default values	Value of start_year	Value of end_year
Yes	Yes	The default value	2021
Yes	No	NULL	2021
No	Yes	The default value	2021
No	No	NULL	2021

Here are some examples:

```
// This example checks if the following three vertices have any outgoing serve edge. The result "Empty set" indicates that such an edge does not exist.
nebula> GO FROM "player666", "player667", "player668" \
    OVER serve \
    YIELD serve.start_year, serve.end_year;
Empty set

nebula> UPSERT EDGE on serve \
    "player666" -> "team200"@0 \
    SET end_year = 2021 \
    WHEN end_year == 2010 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| __NULL__ | 2021 |
+-----+-----+

nebula> UPSERT EDGE on serve \
    "player666" -> "team200"@0 \
    SET end_year = 2022 \
    WHEN end_year == 2010 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| __NULL__ | 2021 |
+-----+-----+

nebula> UPSERT EDGE on serve \
    "player667" -> "team200"@0 \
    SET end_year = 2022 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| __NULL__ | 2022 |
+-----+-----+

nebula> UPSERT EDGE on serve \
    "player668" -> "team200"@0 \
    SET start_year = 2000, end_year = end_year + 1 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| 2000 | __NULL__ |
+-----+-----+
```

In the last query of the preceding example, since `end_year` has no default value, when the edge is created, `end_year` is `NULL`, and `end_year = end_year + 1` does not take effect. But if `end_year` has a default value, `end_year = end_year + 1` will take effect. For example:

```
nebula> CREATE EDGE serve_with_default(start_year int, end_year DEFAULT 2010);
Execution succeeded

nebula> UPSERT EDGE on serve_with_default \
    "player668" -> "team200" \
    SET end_year = end_year + 1 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| __NULL__ | 2011 |
+-----+-----+
```

Update an edge if it exists

If the edge exists and the `WHEN` conditions are met, the edge is updated.

```
nebula> MATCH (v:player{name:"Ben Simmons"})-[e:serve]-(v2) \
    RETURN e;
+-----+
| e
+-----+
| [:serve "player149"->"team219" @0 {end_year: 2019, start_year: 2016}] |
+-----+  
  
nebula> UPSERT EDGE on serve \
    "player149" -> "team219" \
    SET end_year = end_year + 1 \
    WHEN start_year == 2016 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| 2016       | 2020      |
+-----+-----+
```

If the edge exists and the `WHEN` conditions are not met, the update does not take effect.

```
nebula> MATCH (v:player{name:"Ben Simmons"})-[e:serve]-(v2) \
    RETURN e;
+-----+
| e
+-----+
| [:serve "player149"->"team219" @0 {end_year: 2020, start_year: 2016}] |
+-----+  
  
nebula> UPSERT EDGE on serve \
    "player149" -> "team219" \
    SET end_year = end_year + 1 \
    WHEN start_year != 2016 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| 2016       | 2020      |
+-----+-----+
```

Last update: August 17, 2021

4.13 Native index statements

4.13.1 CREATE INDEX

Prerequisites

Before you create an index, make sure that the relative tag or edge type is created. For how to create tags or edge types, see [CREATE TAG](#) and [CREATE EDGE](#).

For how to create full-text indexes, see [Deploy full-text index](#).

Must-read for using indexes

The concept and using restrictions of indexes are comparatively complex. You can use it together with `LOOKUP` and `MATCH` statements.

You can use `CREATE INDEX` to add native indexes for the existing tags, edge types, or properties. They are usually called as tag indexes, edge type indexes, and property indexes.

Tag indexes and edge type indexes apply to queries related to the tag and the edge type, but do not apply to queries that are based on certain properties on the tag. For example, you can use `LOOKUP` to retrieve all the edges with edge type `E`.

Property indexes apply to property-based queries. For example, you can use the `Age` property to retrieve the VID of all vertices that meet `Age == 19`.

If a property index `i_TA` is created for property `A` of tag `T`, there is no need to create an additional tag index `i_T` for tag `T`. This is because the query engine can use `i_TA` to replace `i_T`. The edge type index is the same. However, `i_T` cannot replace `i_TA` for property queries.

Caution

Indexes can dramatically reduce the write performance. The performance reduction can be as much as 90% or even more. **DO NOT** use indexes in production environments unless you are fully aware of their influences on your service.

Indexes cannot make queries faster. It can only locate a vertex or an edge according to properties or count the number of vertices or edges.

Long indexes decrease the scan performance of the Storage Service and use more memory. We suggest that you set the indexing length the same as that of the longest string to be indexed. The longest index length is 255 bytes. Strings longer than 255 bytes will be truncated.

If you must use indexes, we suggest that you:

1. Import the data into Nebula Graph.
2. Create indexes.
3. [Rebuild indexes](#).
4. After the index is created and the data is imported, you can use `LOOKUP` or `MATCH` to retrieve the data. You do not need to specify which indexes to use in a query, Nebula Graph figures that out by itself.

Note

If you create an index before importing the data, the importing speed will be extremely slow due to the reduction in the write performance.

Keep `--disable_auto_compaction = false` during daily incremental writing.

The newly created index will not take effect immediately. Trying to use a newly created index (such as `LOOKUP` or `REBUILD INDEX`) may fail and return `can't find xxx` in the space because the creation is implemented asynchronously. Nebula Graph implements the creation in the next heartbeat cycle. To make sure the creation is successful, take one of the following approaches:

- Find the new index in the result of `SHOW TAG/EDGE INDEXES`.
- Wait for two heartbeat cycles, i.e., 20 seconds. To change the heartbeat interval, modify the `heartbeat_interval_secs` in the [configuration files](#) for all services.

Danger

After creating a new index, or dropping the old index and creating a new one with the same name again, you must `REBUILD INDEX`. Otherwise, these data cannot be returned in the `MATCH` and `LOOKUP` statements.

Syntax

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

Parameter	Description
<code>TAG \ EDGE</code>	Specifies the index type that you want to create.
<code>IF NOT EXISTS</code>	Detects if the index that you want to create exists. If it does not exist, a new one will be created.
<code><index_name></code>	The name of the index. It must be unique in a graph space. A recommended way of naming is <code>i_tagName_propName</code> . The name of the index is case-sensitive and allows letters, numbers, or underscores. Keywords and reserved words are not allowed.
<code><tag_name> \ <edge_name></code>	Specifies the name of the tag or edge associated with the index.
<code><prop_name_list></code>	To index a variable-length string property, you must use <code>prop_name(length)</code> to specify the index length. To index a tag or an edge type, ignore the <code>prop_name_list</code> .
<code>COMMENT</code>	The remarks of the index. The maximum length is 256 bytes. By default, there will be no comments on an index.

Create tag/edge type indexes

```
nebula> CREATE TAG INDEX player_index on player();
nebula> CREATE EDGE INDEX follow_index on follow();
```

After indexing a tag or an edge type, you can use the `LOOKUP` statement to retrieve the VID of all vertices with the tag, or the source vertex ID, destination vertex ID, and ranks of all edges with the edge type. For more information, see [LOOKUP](#).

Create single-property indexes

```
nebula> CREATE TAG INDEX player_index_0 on player(name(10));
```

The preceding example creates an index for the `name` property on all vertices carrying the `player` tag. This example creates an index using the first 10 characters of the `name` property.

```
# To index a variable-length string property, you need to specify the index length.
nebula> CREATE TAG var_string(p1 string);
nebula> CREATE TAG INDEX var ON var_string(p1(10));

# To index a fixed-length string property, you do not need to specify the index length.
nebula> CREATE TAG fix_string(p1 FIXED_STRING(10));
nebula> CREATE TAG INDEX fix ON fix_string(p1);

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

Create composite property indexes

An index on multiple properties on a tag (or an edge type) is called a composite property index.

```
nebula> CREATE TAG INDEX player_index_1 on player(name(10), age);
```

⚠ Caution

Creating composite property indexes across multiple tags or edge types is not supported.

Nebula Graph follows the left matching principle to select indexes when composite property indexes are used in `LOOKUP` or `MATCH` statements. That is, columns in the `WHERE` conditions must be in the first N columns of the index. For example:

```
# This example creates a composite property index for the first 3 properties of tag t.
nebula> CREATE TAG INDEX example_index ON t(p1, p2, p3);

# Note: The index match is not successful because it does not start from p1.
nebula> LOOKUP ON t WHERE p2 == 1 and p3 == 1;

# The index match is successful.
nebula> LOOKUP ON t WHERE p1 == 1;
# The index match is successful because p1 and p2 are consecutive.
nebula> LOOKUP ON t WHERE p1 == 1 and p2 == 1;
# The index match is successful because p1, p2, and p3 are consecutive.
nebula> LOOKUP ON t WHERE p1 == 1 and p2 == 1 and p3 == 1;
```

Last update: August 18, 2021

4.13.2 SHOW INDEXES

`SHOW INDEXES` shows the defined tag or edge type indexes names in the current graph space.

Syntax

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

Examples

```
nebula> SHOW TAG INDEXES;
+-----+-----+-----+
| Index Name | By Tag | Columns |
+-----+-----+-----+
| "fix"      | "fix_string" | ["p1"]   |
+-----+-----+-----+
| "player_index_0" | "player" | ["name"] |
+-----+-----+-----+
| "player_index_1" | "player" | ["name", "age"] |
+-----+-----+-----+
| "var"      | "var_string" | ["p1"]   |
+-----+-----+-----+


nebula> SHOW EDGE INDEXES;
+-----+-----+-----+
| Index Name | By Edge | Columns |
+-----+-----+-----+
| "follow_index" | "follow" | []      |
+-----+-----+-----+
```

Legacy version compatibility

In Nebula Graph 2.0.1, the `SHOW TAG/EDGE INDEXES` statement only returns `Names`.

Last update: August 18, 2021

4.13.3 SHOW CREATE INDEX

`SHOW CREATE INDEX` shows the statement used when creating a tag or an edge type. It contains detailed information about the index, such as its associated properties.

Syntax

```
SHOW CREATE {TAG | EDGE} INDEX <index_name>;
```

Examples

You can run `SHOW TAG INDEXES` to list all tag indexes, and then use `SHOW CREATE TAG INDEX` to show the information about the creation of the specified index.

```
nebula> SHOW TAG INDEXES;
+-----+-----+-----+
| "player_index_0" | "player" | ["name"] |
+-----+-----+-----+
| "player_index_1" | "player" | ["name", "age"] |
+-----+-----+-----+

nebula> SHOW CREATE TAG INDEX player_index_1;
+-----+-----+
| Tag Index Name | Create Tag Index |
+-----+-----+
| "player_index_1" | "CREATE TAG INDEX `player_index_1` ON `player` ( | |
| | `name(20)` | |
| | )" | |
+-----+-----+
```

Edge indexes can be queried through a similar approach.

```
nebula> SHOW EDGE INDEXES;
+-----+-----+-----+
| Index Name | By Edge | Columns |
+-----+-----+-----+
| "follow_index" | "follow" | [] |
+-----+-----+-----+

nebula> SHOW CREATE EDGE INDEX follow_index;
+-----+-----+
| Edge Index Name | Create Edge Index |
+-----+-----+
| "follow_index" | "CREATE EDGE INDEX `follow_index` ON `follow` ( | |
| | )" | |
+-----+-----+
```

Legacy version compatibility

In Nebula Graph 2.0.1, the `SHOW TAG/EDGE INDEXES` statement only returns `Names`.

Last update: August 18, 2021

4.13.4 DESCRIBE INDEX

`DESCRIBE INDEX` can get the information about the index with a given name, including the property name (Field) and the property type (Type) of the index.

Syntax

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

Examples

```
nebula> DESCRIBE TAG INDEX player_index_0;
+-----+-----+
| Field | Type   |
+-----+-----+
| "name" | "fixed_string(30)" |
+-----+-----+
nebula> DESCRIBE TAG INDEX player_index_1;
+-----+-----+
| Field | Type   |
+-----+-----+
| "name" | "fixed_string(10)" |
+-----+-----+
| "age"  | "int64" |
+-----+-----+
```

.....

Last update: August 18, 2021

4.13.5 REBUILD INDEX

Danger

If data is updated or inserted before the creation of the index, you must rebuild the indexes **manually** to make sure that the indexes contain the previously added data. Otherwise, you cannot use `LOOKUP` and `MATCH` to query the data based on the index. If the index is created before any data insertion, there is no need to rebuild the index.

During the rebuilding, all queries skip the index and perform sequential scans. This means that the return results can be different because not all the data is indexed during rebuilding.

You can use `REBUILD INDEX` to rebuild the created tag or edge type index. For details on how to create an index, see [CREATE INDEX](#).

Syntax

```
REBUILD {TAG | EDGE} INDEX [<index_name_list>];
<index_name_list> ::= [index_name [, index_name] ...]
```

- Multiple indexes are permitted in a single `REBUILD` statement, separated by commas. When the index name is not specified, all tag or edge indexes are rebuilt.
- After the rebuilding is complete, you can use the `SHOW {TAG | EDGE} INDEX STATUS` command to check if the index is successfully rebuilt. For details on index status, see [SHOW INDEX STATUS](#).

Examples

```
nebula> CREATE TAG person(name string, age int, gender string, email string);
nebula> CREATE TAG INDEX single_person_index ON person(name(10));

# The following example rebuilds an index and returns the job ID.
nebula> REBUILD TAG INDEX single_person_index;
+-----+
| New Job Id |
+-----+
| 31          |
+-----+

# The following example checks the index status.
nebula> SHOW TAG INDEX STATUS;
+-----+-----+
| Name          | Index Status |
+-----+-----+
| "single_person_index" | "FINISHED"  |
+-----+-----+

# You can also use "SHOW JOB <job_id>" to check if the rebuilding process is complete.
nebula> SHOW JOB 31;
+-----+-----+-----+-----+
| Job Id(TaskId) | Command(Dest) | Status      | Start Time          | Stop Time          |
+-----+-----+-----+-----+
| 31            | "REBUILD_TAG_INDEX" | "FINISHED"  | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:24.000 |
+-----+-----+-----+-----+
| 0             | "storaged1"       | "FINISHED"  | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 |
+-----+-----+-----+-----+
| 1             | "storaged2"       | "FINISHED"  | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 |
+-----+-----+-----+-----+
| 2             | "storaged0"       | "FINISHED"  | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 |
+-----+-----+-----+-----+
```

Nebula Graph creates a job to rebuild the index. The job ID is displayed in the preceding return message. To check if the rebuilding process is complete, use the `SHOW JOB <job_id>` statement. For more information, see [SHOW JOB](#).

Legacy version compatibility

In Nebula Graph 2.x, the `OFFLINE` option is no longer needed or supported.

Last update: August 18, 2021

4.13.6 SHOW INDEX STATUS

`SHOW INDEX STATUS` returns the name of the created tag or edge type index and its status.

The index status includes:

- `QUEUE` : The job is in a queue.
- `RUNNING` : The job is running.
- `FINISHED` : The job is finished.
- `FAILED` : The job has failed.
- `STOPPED` : The job has stopped.
- `INVALID` : The job is invalid.

Note

For details on how to create an index, see [CREATE INDEX](#).

Syntax

```
SHOW {TAG | EDGE} INDEX STATUS;
```

Example

```
nebula> SHOW TAG INDEX STATUS;
+-----+-----+
| Name      | Index Status |
+-----+-----+
| "player_index_0" | "FINISHED" |
+-----+-----+
| "player_index_1" | "FINISHED" |
+-----+-----+
```

Last update: August 18, 2021

4.13.7 DROP INDEX

`DROP INDEX` removes an existing index from the current graph space.

Prerequisite

Running the `DROP INDEX` statement requires some [privileges](#) of `DROP TAG INDEX` and `DROP EDGE INDEX` in the given graph space. Otherwise, Nebula Graph throws an error.

Syntax

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

`IF NOT EXISTS` : Detects whether the index that you want to drop exists. If it exists, it will be dropped.

Example

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

Last update: August 18, 2021

4.14 Full-text index statements

4.14.1 Index overview

Indexes are built to fast process graph queries. Nebula Graph supports two kinds of indexes: native indexes and full-text indexes. This topic introduces the index types and helps choose the right index.

Native indexes

Native indexes allow querying data based on a given property. Features are as follows.

- There are two kinds of native indexes: tag index and edge type index.
- Native indexes must be updated manually. You can use the `REBUILD INDEX` statement to update native indexes.
- Native indexes support indexing multiple properties on a tag or an edge type (composite indexes), but do not support indexing across multiple tags or edge types.
- You can do partial match searches by using composite indexes. The declared fields in the composite index are used from left to right. For more information, see [LOOKUP FAQ](#).
- String operators like `CONTAINS` and `STARTS WITH` are not allowed in `LOOKUP` for native index searching. Use full-text indexes to do fuzzy searches.

OPERATIONS ON NATIVE INDEXES

- [CREATE INDEX](#)
- [SHOW CREATE INDEX](#)
- [SHOW INDEXES](#)
- [DESCRIBE INDEX](#)
- [REBUILD INDEX](#)
- [SHOW INDEX STATUS](#)
- [DROP INDEX](#)
- [LOOKUP](#)
- [MATCH](#)

Full-text indexes

Full-text indexes are used to do prefix, wildcard, regexp, and fuzzy search on a string property. Features are as follows.

- Full-text indexes allow indexing just one property.
- Only strings within a specified length (no longer than 256 bytes) are indexed.
- Full-text indexes do not support logical operations such as `AND`, `OR`, and `NOT`.

Note

To do complete string matches, use native indexes.

OPERATIONS ON FULL-TEXT INDEXES

Before doing any operations on full-text indexes, please make sure that you deploy full-text indexes. Details on full-text indexes deployment, see [Deploy Elasticsearch](#) and [Deploy Listener](#).

At this time, full-text indexes are created automatically on the Elasticsearch cluster. And rebuilding or altering full-text indexes are not supported. To drop full-text indexes, you need to drop them on the Elasticsearch cluster manually.

To query full-text indexes, see [Search with full-text indexes](#).

Null values

Indexes do not support indexing null values.

Range queries

In addition to querying single results from native indexes, you can also do range queries. Not all the native indexes support range queries. You can only do range searches for numeric, date, and time type properties.

Last update: August 18, 2021

4.14.2 Full-text index restrictions

This document holds the restrictions for full-text indexes. Please read the restrictions very carefully before using the full-text indexes. For now, full-text search has the following limitations:

1. The maximum indexing string length is 256 bytes. The part of data that exceeds 256 bytes will not be indexed.
 2. Full-text index can not be applied to more than one property at a time (similar to a composite index).
 3. The `WHERE` clause in full-text search statement `LOOKUP` does not support logical expressions `AND` and `OR`.
 4. Full-text index can not be applied to multiple tags search.
 5. Sorting for the returned results of the full-text search is not supported. Data is returned in the order of data insertion.
 6. Full-text index can not search the null properties.
 7. Rebuilding or altering Elasticsearch indexes is not supported at this time.
 8. Pipe is not supported in the `LOOKUP` statement, excluding the examples in our document.
 9. Full-text search only works on single terms.
 10. Full-text indexes are not deleted together with the graph space.
 11. Make sure that you start the Elasticsearch cluster and Nebula Graph at the same time. If not, the data writing on the Elasticsearch cluster can be incomplete.
 12. Do not contain `'` or `\` in the vertex or edge values. If not, an error is caused in the Elasticsearch cluster storage.
 13. It may take a while for Elasticsearch to create indexes. If Nebula Graph warns no index is found, wait for the index to take effect.
-

Last update: May 7, 2021

4.14.3 Deploy full-text index

Nebula Graph full-text indexes are powered by [Elasticsearch](#). This means that you can use Elasticsearch full-text query language to retrieve what you want. Full-text indexes are managed through built-in procedures. They can be created only for variable `STRING` and `FIXED_STRING` properties when the listener cluster and the Elasticsearch cluster are deployed.

Before you start

Before you start using the full-text index, please make sure that you know the [restrictions](#).

Deploy Elasticsearch cluster

To deploy an Elasticsearch cluster, see the [Elasticsearch documentation](#).

When the Elasticsearch cluster is started, add the template file for the Nebula Graph full-text index. Take the following sample template for example:

```
{
  "template": "nebula*",
  "settings": {
    "index": {
      "number_of_shards": 3,
      "number_of_replicas": 1
    }
  },
  "mappings": {
    "properties" : {
      "tag_id" : { "type" : "long" },
      "column_id" : { "type" : "text" },
      "value" :{ "type" : "keyword"}
    }
  }
}
```

Make sure that you specify the following fields in strict accordance with the preceding template format:

```
"template": "nebula*"
"tag_id" : { "type" : "long" },
"column_id" : { "type" : "text" },
"value" :{ "type" : "keyword"}
```

You can configure the Elasticsearch to meet your business needs. To customize the Elasticsearch, see [Elasticsearch Document](#).

Sign in to the text search clients

```
SIGN IN TEXT SERVICE [(<elastic_ip:port> [,<username>, <password>]), (<elastic_ip:port>), ...]
```

When the Elasticsearch cluster is deployed, use the `SIGN IN` statement to sign in to the Elasticsearch clients. Multiple `elastic_ip:port` pairs are separated with commas. You must use the IPs and the port number in the configuration file for the Elasticsearch. For example:

```
nebula> SIGN IN TEXT SERVICE (127.0.0.1:9200);
```

Elasticsearch does not have username or password by default. If you configured a username and password, you need to specify in the `SIGN IN` statement.

Show text search clients

```
SHOW TEXT SEARCH CLIENTS
```

Use the `SHOW TEXT SEARCH CLIENTS` statement to list the text search clients. For example:

```
nebula> SHOW TEXT SEARCH CLIENTS;
+-----+-----+
| Host      | Port   |
+-----+-----+
```

```
| "127.0.0.1" | 9200 |
+-----+-----+
| "127.0.0.1" | 9200 |
+-----+-----+
| "127.0.0.1" | 9200 |
+-----+-----+
```

Sign out to the text search clients

```
SIGN OUT TEXT SERVICE
```

Use the `SIGN OUT TEXT SERVICE` to sign out all the text search clients. For example:

```
nebula> SIGN OUT TEXT SERVICE;
```

Last update: December 17, 2020

4.14.4 Deploy Raft Listener for Nebula Storage service

Full-Text index data is written to the Elasticsearch cluster asynchronously. The Raft Listener (hereinafter shortened as Listener) is a separate process that fetches data from the Storage Service and writes them into the Elasticsearch cluster.

Prerequisites

- You have read and fully understand the [restrictions](#) for using Full-Text indexes.
- You have [deployed a Nebula Graph cluster](#).
- You have prepared at least one extra Storage Server. To use the Full-Text search, you must run one or more Storage Server as the Raft Listener.

Precautions

- The Storage Service that you want to run as a Listener must have the same or later version with all the other Nebula Graph services in the cluster.
- For now, you can only add Listeners to a graph space once and for all. Trying to add listeners to a graph space that already has a listener will fail. To add multiple listeners, set them [in one statement](#).

Step 1: Prepare the configuration file for the Listeners

You have to prepare a Listener configuration file on the machine that you want to deploy the Listeners. The file name must be `nebula-storaged-listener.conf`. A [template](#) is provided for your reference.

Note

Use real IP addresses in the configuration file instead of domain names or loopback IP addresses such as `127.0.0.1`.

Step 2: Start the Listeners

Run the following command to start the Listeners.

```
./bin/nebula-storaged --flagfile ${listener_config_path}/nebula-storaged-listener.conf
```

`${listener_config_path}` is the path where you store the Listener configuration file.

Step 3: Add Listeners to Nebula Graph

Connect to Nebula Graph and run `USE <space>` to enter the graph space that you want to create Full-Text indexes for. Then run the following statement to add the Listener into Nebula Graph.

Note

You must use real IPs for the listeners.

```
ADD LISTENER ELASTICSEARCH <listener_ip:port> [<listener_ip:port>, ...]
```

Multiple `listener_ip:port` pairs are separated with commas. For example:

```
nebula> ADD LISTENER ELASTICSEARCH 192.168.8.5:46780,192.168.8.6:46780;
```

Show Listeners

Run the `SHOW LISTENER` statement to list the Listeners.

For example:

```
nebula> SHOW LISTENER;
+-----+-----+-----+
| PartId | Type      | Host          | Status   |
+-----+-----+-----+
| 1      | "ELASTICSEARCH" | "[192.168.8.5:46780]" | "ONLINE" |
+-----+-----+-----+
| 2      | "ELASTICSEARCH" | "[192.168.8.5:46780]" | "ONLINE" |
+-----+-----+-----+
| 3      | "ELASTICSEARCH" | "[192.168.8.5:46780]" | "ONLINE" |
+-----+-----+-----+
```

Remove Listeners

Run the `REMOVE LISTENER ELASTICSEARCH` statement to remove all the Elasticsearch Listeners for a graph space.

For example:

```
nebula> REMOVE LISTENER ELASTICSEARCH;
```

What to do next

After deploying the [Elasticsearch cluster](#) and the Listeners, Full-Text indexes are created automatically on the Elasticsearch cluster. You can do Full-Text search now. For more information, see [Full-Text search](#).

Last update: April 22, 2021

4.14.5 Full-text indexes

Full-text indexes are used to do prefix, wildcard, regexp, and fuzzy search on a string property.

You can use the `WHERE` clause to specify the search strings in `LOOKUP` or `MATCH`.

Prerequisite

Before using the full-text index, make sure that you have deployed a Elasticsearch cluster and a Listener cluster. For more information, see [Deploy Elasticsearch](#) and [Deploy Listener](#).

Precaution

Before using the full-text index, make sure that you know the [restrictions](#).

Natural language full-text search

A natural language search interprets the search string as a phrase in natural human language. The search is case-insensitive. By default, each substring (separated by spaces) will be searched separately. For example, there are three vertices with the tag `player`. The tag `player` contains the property `name`. The `name` of these three vertices are `Kevin Durant`, `Tim Duncan`, and `David Beckham`. Now that the full-text index of `player.name` is established, these three vertices will be queried when using the prefix search statement `LOOKUP ON player WHERE PREFIX(player.name, "d")`.

Syntax

Create full-text indexes

```
CREATE {TAG | EDGE} INDEX <index_name> ON {<tag_name> | <edge_name>} ([<prop_name_list>]);
```

Show full-text indexes

```
SHOW FULLTEXT INDEXES;
```

REBUILD FULL-TEXT INDEXES

```
REBUILD FULLTEXT INDEX;
```

DROP FULL-TEXT INDEXES

```
DROP FULLTEXT INDEX <index_name>;
```

USE QUERY OPTIONS

```
LOOKUP ON {<tag> | <edge_type>} WHERE <expression> [YIELD <return_list>];

<expression> ::=  
PREFIX | WILDCARD | REGEXP | FUZZY
```

```
<return_list>
  <prop_name> [AS <prop_alias>] [, <prop_name> [AS <prop_alias>] ...]
```

- PREFIX(schema_name.prop_name, prefix_string, row_limit, timeout)
- WILDCARD(schema_name.prop_name, wildcard_string, row_limit, timeout)
- REGEXP(schema_name.prop_name, regexp_string, row_limit, timeout)
- FUZZY(schema_name.prop_name, fuzzy_string, fuzziness, operator, row_limit, timeout)
 - fuzziness (optional): Maximum edit distance allowed for matching. The default value is `AUTO`. For other valid values and more information, see [Elasticsearch document](#).
 - operator (optional): Boolean logic used to interpret the text. Valid values are `OR` (default) and `AND`.
- row_limit (optional): Specifies the number of rows to return. The default value is `100`.
- timeout (optional): Specifies the timeout time. The default value is `200ms`.

Examples

```
// This example creates the graph space.
nebula> CREATE SPACE basketballplayer (partition_num=3,replica_factor=1, vid_type=fixed_string(30));

// This example signs in the text service.
nebula> SIGN IN TEXT SERVICE (127.0.0.1:9200);

// This example switches the graph space.
nebula> USE basketballplayer;

// This example adds the listener to the Nebula Graph cluster.
nebula> ADD LISTENER ELASTICSEARCH 192.168.8.5:9789;

// This example creates the tag.
nebula> CREATE TAG player(name string, age int);

// This example creates the native index.
nebula> CREATE TAG INDEX name ON player(name(20));

// This example rebuilds the native index.
nebula> REBUILD TAG INDEX;

// This example creates the full-text index. The index name starts with "nebula".
nebula> CREATE FULLTEXT TAG INDEX nebula_index_1 ON player(name);

// This example rebuilds the full-text index.
nebula> REBUILD FULLTEXT INDEX;

// This example shows the full-text index.
nebula> SHOW FULLTEXT INDEXES;
+-----+-----+-----+-----+
| Name      | Schema Type | Schema Name | Fields |
+-----+-----+-----+-----+
| "nebula_index_1" | "Tag"      | "player"    | "name"  |
+-----+-----+-----+-----+

// This example inserts the test data.
nebula> INSERT VERTEX player(name, age) VALUES \
  "Russell Westbrook": ("Russell Westbrook", 30), \
  "Chris Paul": ("Chris Paul", 33), \
  "Boris Diaw": ("Boris Diaw", 36), \
  "David West": ("David West", 38), \
  "Danny Green": ("Danny Green", 31), \
  "Tim Duncan": ("Tim Duncan", 42), \
  "James Harden": ("James Harden", 29), \
  "Tony Parker": ("Tony Parker", 36), \
  "Aron Baynes": ("Aron Baynes", 32), \
  "Ben Simmons": ("Ben Simmons", 22), \
  "Blake Griffin": ("Blake Griffin", 30);

// These examples run test queries.
nebula> LOOKUP ON player WHERE PREFIX(player.name, "B");
+-----+
| _vid      |
+-----+
| "Boris Diaw" |
+-----+
| "Ben Simmons" |
+-----+
| "Blake Griffin" |
+-----+

nebula> LOOKUP ON player WHERE WILDCARD(player.name, "*ri*") YIELD player.name, player.age;
+-----+-----+
| _vid      | name      | age      |
+-----+-----+
```

```

+-----+-----+-----+
| "Chris Paul" | "Chris Paul" | 33 |
+-----+-----+-----+
| "Boris Diaw" | "Boris Diaw" | 36 |
+-----+-----+-----+
| "Blake Griffin" | "Blake Griffin" | 30 |
+-----+-----+-----+
nebula> LOOKUP ON player WHERE WILDCARD(player.name, "*ri*") | YIELD count(*);
+-----+
| count(*) |
+-----+
| 3 |
+-----+
nebula> LOOKUP ON player WHERE REGEXP(player.name, "R.*") YIELD player.name, player.age;
+-----+-----+-----+
| _vid | name | age |
+-----+-----+-----+
| "Russell Westbrook" | "Russell Westbrook" | 30 |
+-----+-----+-----+
nebula> LOOKUP ON player WHERE REGEXP(player.name, ".*");
+-----+
| _vid |
+-----+
| "Danny Green" |
+-----+
| "David West" |
+-----+
| "Russell Westbrook" |
+-----+
...
nebula> LOOKUP ON player WHERE FUZZY(player.name, "Tim Dunncan", AUTO, OR) YIELD player.name;
+-----+-----+
| _vid | name |
+-----+-----+
| "Tim Duncan" | "Tim Duncan" |
+-----+-----+
// This example drops the full-text index.
nebula> DROP FULLTEXT INDEX nebula_index_1;

```

Last update: August 23, 2021

4.15 Subgraph and path

4.15.1 GET SUBGRAPH

The `GET SUBGRAPH` statement retrieves information of vertices and edges reachable from the source vertices of the specified edge types and returns information of the subgraph.

Syntax

```
GET SUBGRAPH [WITH PROP] [<step_count> STEPS] FROM {<vid>, <vid>...}
[IN <edge_type>, <edge_type>...]
[OUT <edge_type>, <edge_type>...]
[BOTH <edge_type>, <edge_type>...];
```

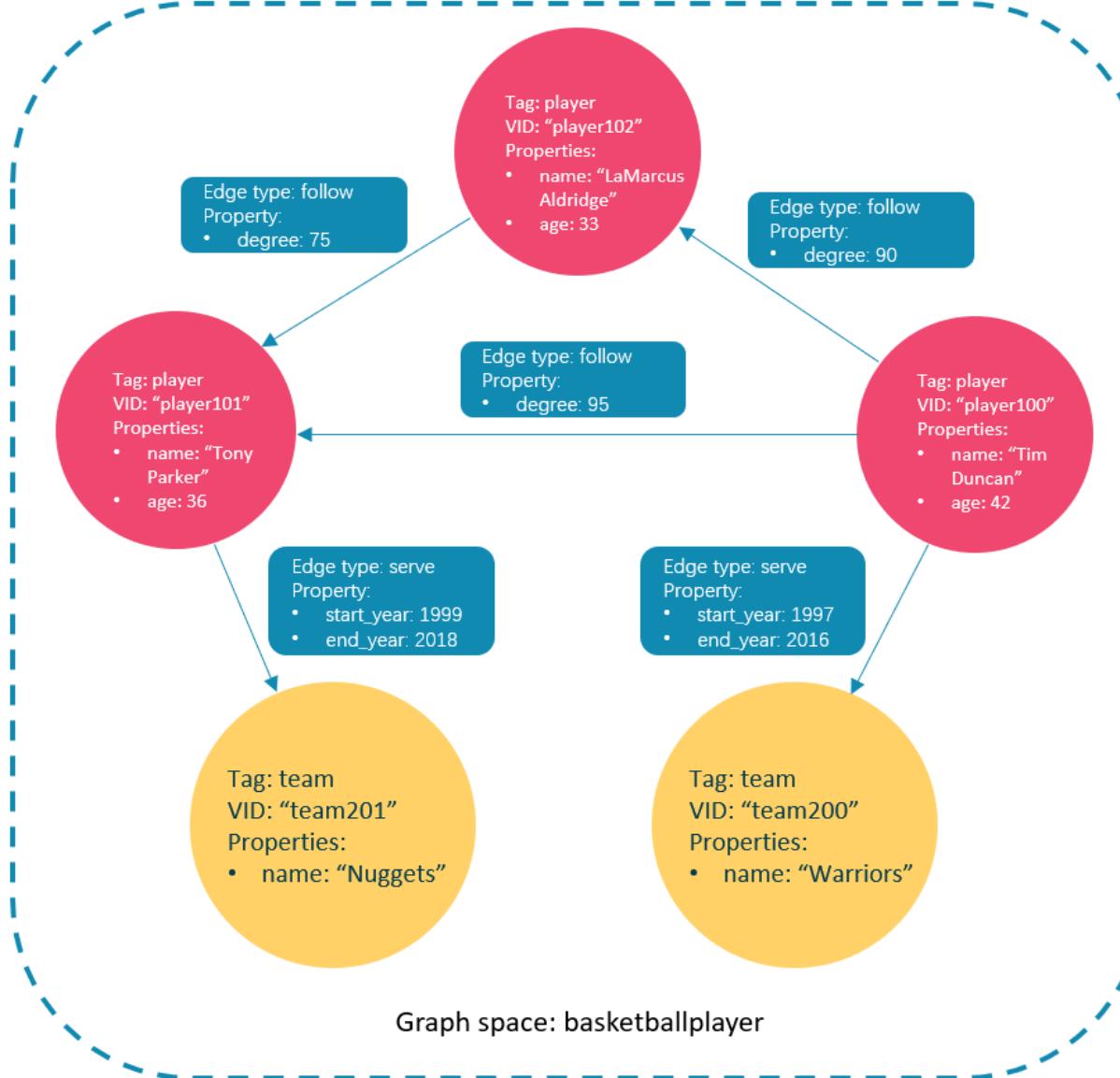
- `WITH PROP` shows the properties of edges. If not specified, the properties will be hidden.
- `step_count` specifies the number of hops from the source vertices and returns the subgraph from 0 to `step_count` hops. It must be a non-negative integer. Its default value is 1.
- `vid` specifies the vertex IDs.
- `edge_type` specifies the edge type. You can use `IN`, `OUT`, and `BOTH` to specify the traversal direction of the edge type. The default is `BOTH`.

Note

The path type of `GET SUBGRAPH` is `trail`. Only vertices can be repeatedly visited in graph traversal. For more information, see [Path](#).

Examples

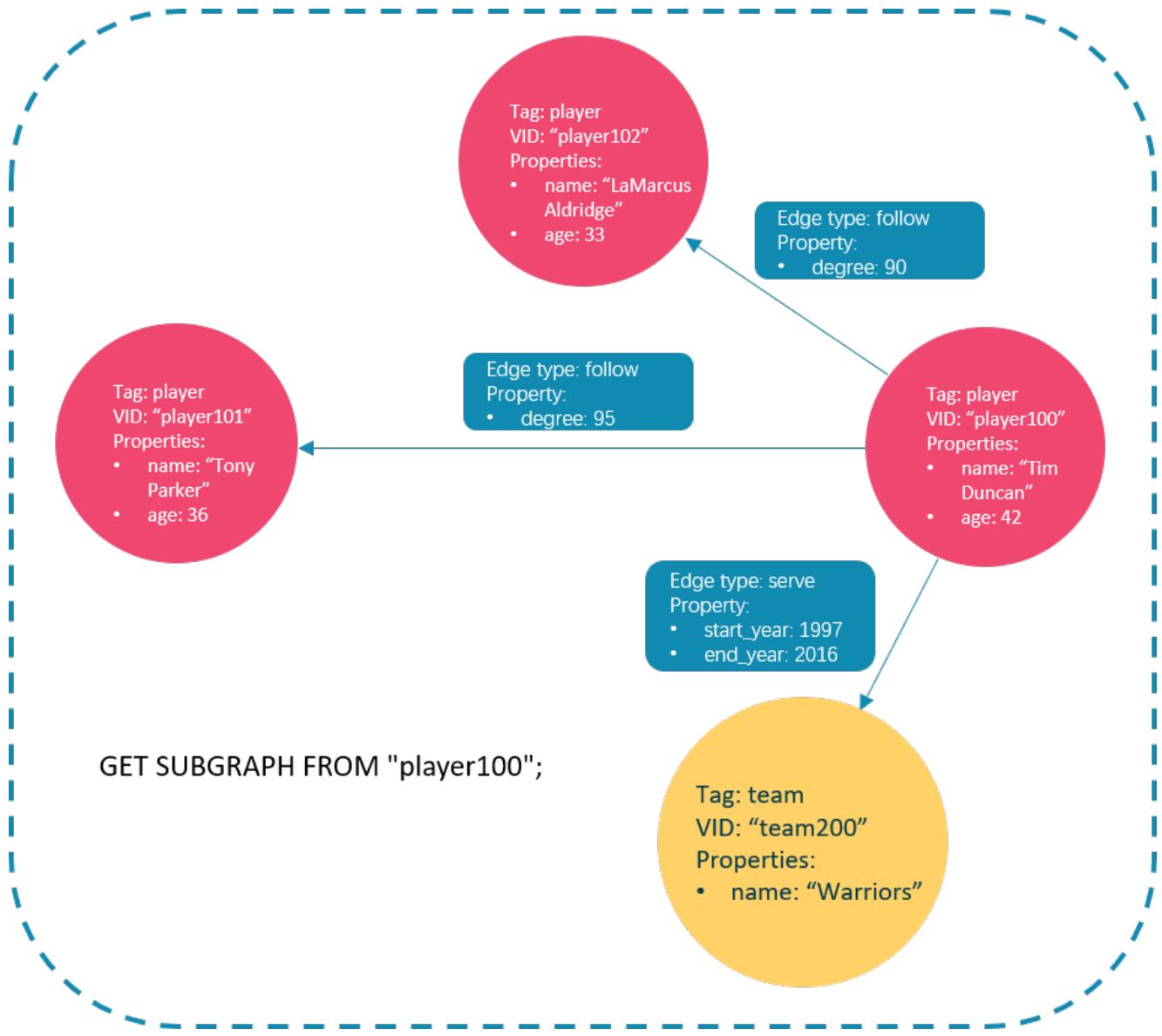
The following graph is used as the sample.



- This example goes one step from the vertex `player100` over all edge types and gets the subgraph.

```
nebula> GET SUBGRAPH 1 STEPS FROM "player100";
+-----+
| _vertices
| _edges
+-----+
| [("player100" :player{age: 42, name: "Tim Duncan"})]
| [[:follow "player100"->"player101" @0 {}], [:follow "player100"->"player102" @0 {}], [:serve "player100"->"team200" @0 {}]] |
+-----+
| [("player102" :player{age: 33, name: "LaMarcus Aldridge"}), ("player101" :player{age: 36, name: "Tony Parker"}), ("team200" :team{name: "Warriors"})]
| [[:follow "player102"->"player101" @0 {}]]
+-----+
```

The returned subgraph is as follows.



- This example goes one step from the vertex `player100` over incoming `follow` edges and gets the subgraph.

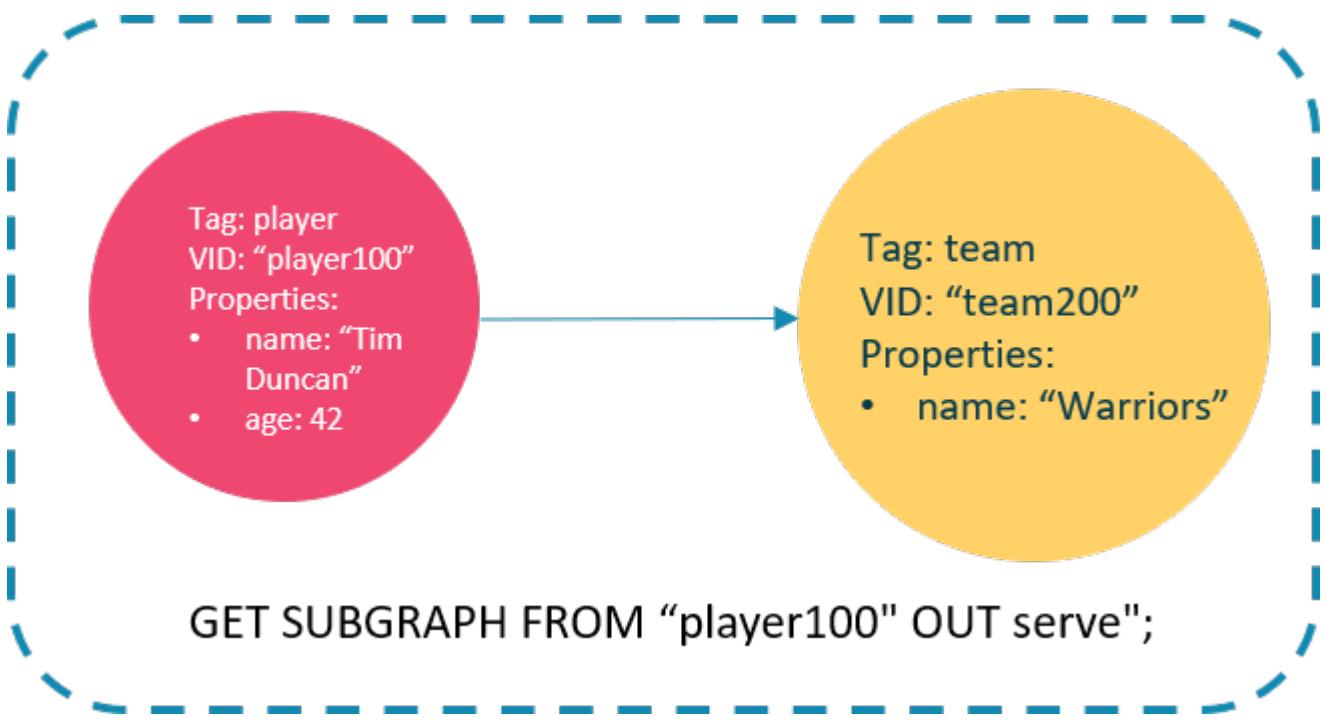
```
nebula> GET SUBGRAPH 1 STEPS FROM "player100" IN follow;
+-----+-----+
| _vertices | _edges |
+-----+-----+
| [{"player100":player{age: 42, name: "Tim Duncan"}}, []] | []
+-----+-----+
| [] | []
+-----+-----+
```

There is no incoming `follow` edge to `player100`, so no vertex or edge is returned.

- This example goes one step from the vertex `player100` over outgoing `serve` edges, gets the subgraph, and shows the property of the edge.

```
nebula> GET SUBGRAPH WITH PROP 1 STEPS FROM "player100" OUT serve;
+-----+-----+
| _vertices | _edges |
+-----+-----+
| [{"player100":player{age: 42, name: "Tim Duncan"}}, [{"serve": "player100->team200", "@0": {end_year: 2016, start_year: 1997}}], []] | []
+-----+-----+
| [{"team200":team{name: "Warriors"}}, []] | []
+-----+-----+
```

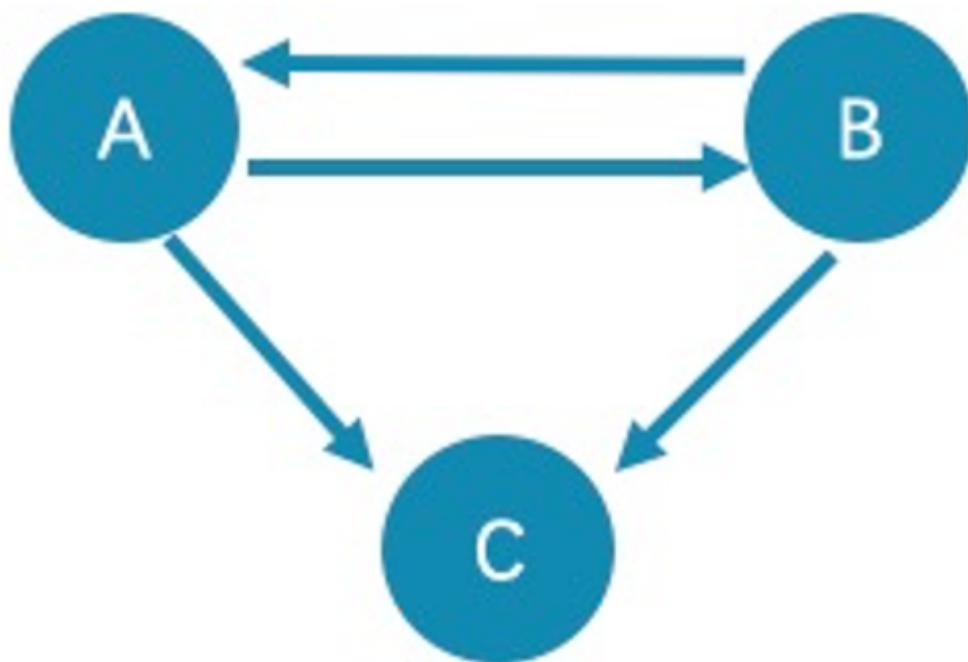
The returned subgraph is as follows.



FAQ

WHY IS THE NUMBER OF HOPS IN THE RETURNED RESULT GREATER THAN STEP_COUNT?

To show the completeness of the subgraph, an additional hop is made on all vertices that meet the conditions. The following graph is used as the sample.



- The returned paths of `GET SUBGRAPH 1 STEPS FROM "A";` are `A->B`, `B->A`, and `A->C`. To show the completeness of the subgraph, an additional hop is made on all vertices that meet the conditions, namely `B->C`.
- The returned path of `GET SUBGRAPH 1 STEPS FROM "A" IN follow;` is `B->A`. To show the completeness of the subgraph, an additional hop is made on all vertices that meet the conditions, namely `A->B`.

If you only query paths or vertices that meet the conditions, we suggest you use `MATCH` or `GO`. The example is as follows.

```
nebula> match p= (v:player) -- (v2) where id(v)=="A" return p;
nebula> go 1 steps from "A" over follow;
```

WHY IS THE NUMBER OF HOPS IN THE RETURNED RESULT LOWER THAN STEP_COUNT?

The query stops when there is not enough subgraph data and will not return the null value.

```
nebula> GET SUBGRAPH 100 STEPS FROM "player141" OUT follow;
+-----+-----+-----+
| _vertices | _edges |
+-----+-----+
| [{"player141" :player{age: 43, name: "Ray Allen"} }] | [[:follow "player141"->"player124" @0 {degree: 9}]] |
+-----+-----+
| [{"player124" :player{age: 33, name: "Rajon Rondo"} }] | [[:follow "player124"->"player141" @0 {degree: -1}]] |
+-----+-----+
```

Last update: August 19, 2021

4.15.2 FIND PATH

The `FIND PATH` statement finds the paths between the selected source vertices and destination vertices.

Syntax

```

FIND { SHORTEST | ALL | NOLOOP } PATH [WITH PROP] FROM <vertex_id_list> TO <vertex_id_list>
OVER <edge_type_list> [REVERSELY | BIDIRECT] [<WHERE clause>] [UPTO <N> STEPS] [| ORDER BY $-.path] [| LIMIT <M>];

<vertex_id_list> ::= 
  [vertex_id [, vertex_id] ...]

```

- `SHORTEST` finds the shortest path.
- `ALL` finds all the paths.
- `NOLOOP` finds the paths without circles.
- `WITH PROP` shows properties of vertices and edges. If not specified, properties will be hidden.
- `<vertex_id_list>` is a list of vertex IDs separated with commas (,). It supports `$-` and `$var`.
- `<edge_type_list>` is a list of edge types separated with commas (,). `*` is all edge types.
- `REVERSELY | BIDIRECT` specifies the direction. `REVERSELY` is reverse graph traversal while `BIDIRECT` is bidirectional graph traversal.
- `<WHERE clause>` filters properties of edges.
- `<N>` is the maximum hop number of the path. The default value is `5`.
- `<M>` specifies the maximum number of rows to return.

Note

The path type of `FIND PATH` is `trail`. Only vertices can be repeatedly visited in graph traversal. For more information, see [Path](#).

Limitations

- When a list of source and/or destination vertex IDs are specified, the paths between any source vertices and the destination vertices will be returned.
- There can be cycles when searching all paths.
- `FIND PATH` only supports filtering properties of edges with `WHERE` clauses. Filtering properties of vertices and functions are not supported for now.
- `FIND PATH` is a single-thread procedure, so it uses much memory.

Examples

A returned path is like `(<vertex_id>)-[:<edge_type_name>@<rank>]->(<vertex_id>)`.

```

nebula> FIND SHORTEST PATH FROM "player102" TO "team204" OVER *;
+-----+
| path
+-----+
| <"player102">-[:serve@0 {}]->("team204") |
+-----+

```

```

nebula> FIND SHORTEST PATH WITH PROP FROM "team204" TO "player100" OVER * REVERSELY;
+-----+
| path
+-----+

```

```
| <("team204" :team{name: "Spurs"})->[:serve@0 {end_year: 2016, start_year: 1997}]-("player100" :player{age: 42, name: "Tim Duncan"})> |  
+-----+  
  
nebula> FIND ALL PATH FROM "player100" TO "team204" OVER * WHERE follow.degree is EMPTY or follow.degree >=0;  
+-----+  
| path |  
+-----+  
| <"player100"[:serve@0 {}]->"team204"> |  
+-----+  
| <"player100"[:follow@0 {}]->"player125"[:serve@0 {}]->"team204"> |  
+-----+  
| <"player100"[:follow@0 {}]->"player101"[:serve@0 {}]->"team204"> |  
+-----+  
...  
  
nebula> FIND NOLOOP PATH FROM "player100" TO "team204" OVER *;  
+-----+  
| path |  
+-----+  
| <"player100"[:serve@0 {}]->"team204"> |  
+-----+  
| <"player100"[:follow@0 {}]->"player125"[:serve@0 {}]->"team204"> |  
+-----+  
| <"player100"[:follow@0 {}]->"player101"[:serve@0 {}]->"team204"> |  
+-----+  
| <"player100"[:follow@0 {}]->"player101"[:follow@0 {}]->"player125"[:serve@0 {}]->"team204"> |  
+-----+  
| <"player100"[:follow@0 {}]->"player101"[:follow@0 {}]->"player102"[:serve@0 {}]->"team204"> |  
+-----+
```

FAQ

DOES IT SUPPORT THE WHERE CLAUSE TO ACHIEVE CONDITIONAL FILTERING DURING GRAPH TRAVERSAL?

`FIND PATH` only supports filtering properties of edges with `WHERE` clauses, such as `FIND ALL PATH FROM "player100" TO "team204"`
`OVER * WHERE follow.degree is EMPTY or follow.degree >=0;`.

Filtering properties of vertices is not supported for now.

Last update: August 19, 2021

4.16 Query tuning statements

4.16.1 EXPLAIN and PROFILE

`EXPLAIN` helps output the execution plan of an nGQL statement without executing the statement.

`PROFILE` executes the statement, then outputs the execution plan as well as the execution profile. You can optimize the queries for better performance according to the execution plan and profile.

Execution Plan

The execution plan is determined by the execution planner in the Nebula Graph query engine.

The execution planner processes the parsed nGQL statements into `actions`. An `action` is the smallest unit that can be executed. A typical `action` fetches all neighbors of a given vertex, gets the properties of an edge, and filters vertices or edges based on the given conditions. Each `action` is assigned to an `operator` that performs the action.

For example, a `SHOW TAGS` statement is processed into two `actions` and assigned to a `Start` operator and a `ShowTags` operator, while a more complex `GO` statement may be processed into more than 10 `actions` and assigned to 10 operators.

Syntax

- `EXPLAIN`

```
EXPLAIN [format="row" | "dot"] <your_nGQL_statement>;
```

- `PROFILE`

```
PROFILE [format="row" | "dot"] <your_nGQL_statement>;
```

Output formats

The output of an `EXPLAIN` or a `PROFILE` statement has two formats, the default `row` format and the `dot` format. You can use the `format` option to modify the output format. Omitting the `format` option indicates using the default `row` format.

The row format

The `row` format outputs the return message in a table as follows.

- EXPLAIN

```
nebula> EXPLAIN format="row" SHOW TAGS;
Execution succeeded (time spent 327/892 us)

Execution Plan

-----+-----+-----+-----+
| id | name      | dependencies | profiling data | operator info
-----+-----+-----+-----+
| 1 | ShowTags | 0           |           | outputVar: [{"colNames":[], "name": "__ShowTags_1", "type": "DATASET"}] |
|   |           |           |           | inputVar:
-----+-----+-----+-----+
| 0 | Start    |           |           | outputVar: [{"colNames":[], "name": "__Start_0", "type": "DATASET"}] |
-----+-----+-----+-----+
```

- PROFILE

```
nebula> PROFILE format="row" SHOW TAGS;
+-----+
| Name   |
+-----+
| player |
+-----+
| team   |
+-----+
Got 2 rows (time spent 2038/2728 us)

Execution Plan

-----+-----+-----+-----+
| id | name      | dependencies | profiling data           | operator
info
-----+-----+-----+-----+
+-----+
| 1 | ShowTags | 0           | ver: 0, rows: 1, execTime: 42us, totalTime: 1177us | outputVar: [{"colNames": [], "name": "__ShowTags_1", "type": "DATASET"}] |
|   |           |           |           | inputVar:
-----+-----+-----+-----+
+-----+
| 0 | Start    |           | ver: 0, rows: 0, execTime: 1us, totalTime: 57us   | outputVar: [{"colNames": [], "name": "__Start_0", "type": "DATASET"}] |
-----+-----+-----+-----+
```

The descriptions are as follows.

Parameter	Description
<code>id</code>	The ID of the <code>operator</code> .
<code>name</code>	The name of the <code>operator</code> .
<code>dependencies</code>	The ID of the <code>operator</code> that the current <code>operator</code> depends on.
<code>profiling</code> <code>data</code>	The content of the execution profile. <code>ver</code> is the version of the <code>operator</code> . <code>rows</code> shows the number of rows to be output by the <code>operator</code> . <code>execTime</code> shows the execution time of <code>action</code> . <code>totalTime</code> is the sum of the execution time, the system scheduling time, and the queueing time.
<code>operator info</code>	The detailed information of the <code>operator</code> .

The dot format

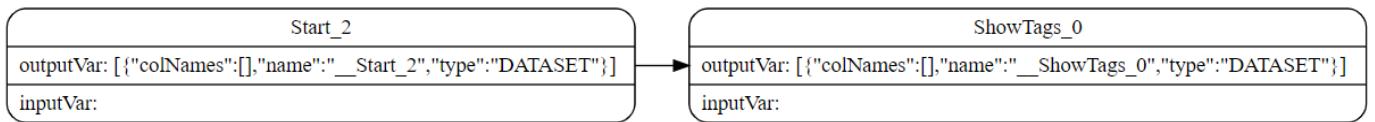
You can use the `format="dot"` option to output the return message in the `dot` language, and then use Graphviz to generate a graph of the plan.

Note

Graphviz is open source graph visualization software. Graphviz provides an online tool for previewing DOT language files and exporting them to other formats such as SVG or JSON. For more information, see [Graphviz Online](#).

```
nebula> EXPLAIN format="dot" SHOW TAGS;
Execution succeeded (time spent 161/665 us)
Execution Plan
-----
plan
-----
graph LR
    Start_2[Start_2  
outputVar: [{"colNames":[],"name":"_Start_2","type":"DATASET"}]] --> ShowTags_0[ShowTags_0  
outputVar: [{"colNames":[],"name":"_ShowTags_0","type":"DATASET"}]]  
    inputVar:
}
-----
```

The Graphviz graph transformed from the above DOT statement is as follows.



Last update: August 24, 2021

4.17 Operation and maintenance statements

4.17.1 BALANCE syntax

The `BALANCE` statements support the load balancing operations of the Nebula Graph Storage services. For more information about storage load balancing and examples for using the `BALANCE` statements, see [Storage load balance](#).

The `BALANCE` statements are listed as follows.

Syntax	Description
<code>BALANCE DATA</code>	Starts a task to balance the distribution of storage partitions in a Nebula Graph cluster. It returns the task ID (<code>balance_id</code>).
<code>BALANCE DATA <balance_id></code>	Shows the status of the <code>BALANCE DATA</code> task.
<code>BALANCE DATA STOP</code>	Stops the <code>BALANCE DATA</code> task.
<code>BALANCE DATA REMOVE <host_list></code>	Scales in the Nebula Graph cluster and detaches specific storage hosts.
<code>BALANCE LEADER</code>	Balances the distribution of storage raft leaders in a Nebula Graph cluster.

Last update: August 23, 2021

4.17.2 Job manager and the JOB statements

The long-term tasks run by the Storage Service are called jobs, such as `COMPACT`, `FLUSH`, and `STATS`. These jobs can be time-consuming if the data amount in the graph space is large. The job manager helps you run, show, stop, and recover jobs.

SUBMIT JOB COMPACT

The `SUBMIT JOB COMPACT` statement triggers the long-term RocksDB `compact` operation.

For more information about `compact` configuration, see [Storage Service configuration](#).

EXAMPLE

```
nebula> SUBMIT JOB COMPACT;
+-----+
| New Job Id |
+-----+
| 40          |
+-----+
```

SUBMIT JOB FLUSH

The `SUBMIT JOB FLUSH` statement writes the RocksDB memfile in the memory to the hard disk.

EXAMPLE

```
nebula> SUBMIT JOB FLUSH;
+-----+
| New Job Id |
+-----+
| 96          |
+-----+
```

SUBMIT JOB STATS

The `SUBMIT JOB STATS` statement starts a job that makes the statistics of the current graph space. Once this job succeeds, you can use the `SHOW STATS` statement to list the statistics. For more information, see [SHOW STATS](#).

Note

If the data stored in the graph space changes, in order to get the latest statistics, you have to run `SUBMIT JOB STATS` again.

EXAMPLE

```
nebula> SUBMIT JOB STATS;
+-----+
| New Job Id |
+-----+
| 97          |
+-----+
```

SHOW JOB

The Meta Service parses a `SUBMIT JOB` request into multiple tasks and assigns them to the nebula-storaged processes. The `SHOW JOB <job_id>` statement shows the information about a specific job and all its tasks.

`job_id` is returned when you run the `SUBMIT JOB` statement.

EXAMPLE

```
nebula> SHOW JOB 96;
+-----+-----+-----+-----+-----+
| Job Id(LinkId) | Command(Dest) | Status      | Start Time           | Stop Time           |
+-----+-----+-----+-----+-----+
| 96           | "FLUSH"      | "FINISHED"  | 2020-11-28T14:14:29.000 | 2020-11-28T14:14:29.000 |
+-----+-----+-----+-----+-----+
```

0	"storaged2"	"FINISHED"	2020-11-28T14:14:29.000	2020-11-28T14:14:29.000
1	"storaged0"	"FINISHED"	2020-11-28T14:14:29.000	2020-11-28T14:14:29.000
2	"storaged1"	"FINISHED"	2020-11-28T14:14:29.000	2020-11-28T14:14:29.000

The descriptions are as follows.

Parameter	Description
Job Id(TaskId)	The first row shows the job ID and the other rows show the task IDs.
Command(Dest)	The first row shows the command executed and the other rows show on which storaged processes the task is running.
Status	Shows the status of the job or task. For more information, see Job status .
Start Time	Shows a timestamp indicating the time when the job or task enters the <code>RUNNING</code> phase.
Stop Time	Shows a timestamp indicating the time when the job or task gets <code>FINISHED</code> , <code>FAILED</code> , or <code>STOPPED</code> .

JOB STATUS

The descriptions are as follows.

Status	Description
QUEUE	The job or task is waiting in a queue. The <code>Start Time</code> is empty in this phase.
RUNNING	The job or task is running. The <code>Start Time</code> shows the beginning time of this phase.
FINISHED	The job or task is successfully finished. The <code>Stop Time</code> shows the time when the job or task enters this phase.
FAILED	The job or task has failed. The <code>Stop Time</code> shows the time when the job or task enters this phase.
STOPPED	The job or task is stopped without running. The <code>Stop Time</code> shows the time when the job or task enters this phase.
REMOVED	The job or task is removed.

The description of switching the status is described as follows.

```
Queue -- running -- finished -- removed
  \       \   /
  \       \ -- failed -- /
  \       \   /
  \ ----- stopped --/
```

SHOW JOBS

The `SHOW JOBS` statement lists all the unexpired jobs.

The default job expiration interval is one week. You can change it by modifying the `job_expired_secs` parameter of the Meta Service. For how to modify `job_expired_secs`, see [Meta Service configuration](#).

EXAMPLE

nebula> SHOW JOBS;				
Job Id	Command	Status	Start Time	Stop Time
97	"STATS"	"FINISHED"	2020-11-28T14:48:52.000	2020-11-28T14:48:52.000
96	"FLUSH"	"FINISHED"	2020-11-28T14:14:29.000	2020-11-28T14:14:29.000
95	"STATS"	"FINISHED"	2020-11-28T13:02:11.000	2020-11-28T13:02:11.000
86	"REBUILD_EDGE_INDEX"	"FINISHED"	2020-11-26T13:38:24.000	2020-11-26T13:38:24.000

STOP JOB

The `STOP JOB` statement stops jobs that are not finished.

EXAMPLE

```
nebula> STOP JOB 22;
+-----+
| Result      |
+-----+
| "Job stopped" |
+-----+
```

RECOVER JOB

The `RECOVER JOB` statement re-executes the failed jobs and returns the number of recovered jobs.

EXAMPLE

```
nebula> RECOVER JOB;
+-----+
| Recovered job num |
+-----+
| 5 job recovered   |
+-----+
```

FAQ

HOW TO TROUBLESHOOT JOB PROBLEMS?

The `SUBMIT JOB` operations use the HTTP port. Please check if the HTTP ports on the machines where the Storage Service is running are working well. You can use the following command to debug.

```
curl "http://{storage-ip}:19779/admin?space={space_name}&op=compact"
```

Last update: August 23, 2021

5. Deployment and installation

5.1 Prepare resources for compiling, installing, and running Nebula Graph

This topic describes the requirements and suggestions for compiling and installing Nebula Graph, as well as how to estimate the resource you need to reserve for running a Nebula Graph cluster.

5.1.1 Reading guide

If you are reading this topic with the questions listed below, click them to jump to their answers.

- [What do I need to compile Nebula Graph?](#)
- [What do I need to run Nebula Graph in a test environment?](#)
- [What do I need to run Nebula Graph in a production environment?](#)
- [How much memory and disk space do I need to reserve for my Nebula Graph cluster?](#)

5.1.2 Requirements for compiling the Nebula Graph source code

Hardware requirements for compiling Nebula Graph

Item	Requirement
CPU architecture	x86_64
Memory	4 GB
Disk	10 GB, SSD

Supported operating systems for compiling Nebula Graph

For now, we can only compile Nebula Graph in the Linux system. We recommend that you use any Linux system with kernel version 2.6.32 or above.

Software requirements for compiling Nebula Graph

You must have the correct version of the software listed below to compile Nebula Graph. If they are not as required or you are not sure, follow the steps in [Prepare software for compiling Nebula Graph](#) to get them ready.

Software	Version	Note
glibc	2.17 or above	You can run <code>ldd --version</code> to check the glibc version.
make	Any stable version	-
m4	Any stable version	-
git	Any stable version	-
wget	Any stable version	-
unzip	Any stable version	-
xz	Any stable version	-
readline-devel	Any stable version	-
ncurses-devel	Any stable version	-
zlib-devel	Any stable version	-
gcc	7.5.0 or above	You can run <code>gcc -v</code> to check the gcc version.
gcc-c++	Any stable version	-
cmake	3.9.0 or above	You can run <code>cmake --version</code> to check the cmake version.
gettext	Any stable version	-
curl	Any stable version	-
redhat-lsb-core	Any stable version	-
libstdc++-static	Any stable version	Only needed in CentOS 8+, RedHat 8+, and Fedora systems.
libasan	Any stable version	Only needed in CentOS 8+, RedHat 8+, and Fedora systems.
bzip2	Any stable version	-

Other third-party software will be automatically downloaded and installed to the `build` directory at the configure (cmake) stage.

Prepare software for compiling Nebula Graph

This section guides you through the downloading and installation of software required for compiling Nebula Graph.

1. Install dependencies.

- For CentOS, RedHat, and Fedora users, run the following commands.

```
$ 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 \
  bzip2
// For CentOS 8+, RedHat 8+, and Fedora, install libstdc++-static and libasan as well
$ yum install -y libstdc++-static libasan
```

- For Debian and Ubuntu users, run the following commands.

```
$ 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
```

2. Check if the GCC and cmake on your host are in the right version. See [Software requirements for compiling Nebula Graph](#) for the required versions.

```
$ g++ --version
$ cmake --version
```

If your GCC and CMake are in the right version, then you are all set. If they are not, follow the sub-steps as follows.

1. Clone the `nebula-common` repository to your host.

```
```bash
$ git clone -b master https://github.com/vesoft-inc/nebula-common.git
```
Users can use the `--branch` or `-b` option to specify the branch to be cloned. For example, for 2.5.0, run the following command.
```bash
$ git clone --branch v2.5.0 https://github.com/vesoft-inc/nebula-common.git
```

```

2. Make `nebula-common` the current working directory.

```
```bash
$ cd nebula-common
```

```

3. Run the following commands to install and enable CMake and GCC.

```
```bash
// Install CMake.
$./third-party/install-cmake.sh cmake-install

// Enable CMake.
$ source cmake-install/bin/enable-cmake.sh

// Authorize the write privilege to the opt directory.
$ sudo mkdir /opt/vesoft && sudo chmod -R a+w /opt/vesoft

// Install GCC. Installing GCC to the opt directory requires the write privilege. And users can change it to other locations.
$./third-party/install-gcc.sh --prefix=/opt

// Enable GCC.
$ source /opt/vesoft/toolset/gcc/7.5.0/enable
```

```

3. Execute the script `install-third-party.sh`.

```
$ ./third-party/install-third-party.sh
```

5.1.3 Requirements and suggestions for installing Nebula Graph in test environments

Hardware requirements for test environments

| Item | Requirement |
|--------------------|-------------|
| CPU architecture | x86_64 |
| Number of CPU core | 4 |
| Memory | 8 GB |
| Disk | 100 GB, SSD |

Supported operating systems for test environments

For now, we can only install Nebula Graph in the Linux system. To install Nebula Graph in a test environment, we recommend that you use any Linux system with kernel version 3.9 or above.

Suggested service architecture for test environments

| Process | Suggested number |
|---|------------------|
| metad (the metadata service process) | 1 |
| storaged (the storage service process) | 1 or more |
| graphd (the query engine service process) | 1 or more |

For example, for a single-machine test environment, you can deploy 1 metad, 1 storaged, and 1 graphd processes in the machine.

For a more common test environment, such as a cluster of 3 machines (named as A, B, and C), you can deploy Nebula Graph as follows:

| Machine name | Number of metad | Number of storaged | Number of graphd |
|--------------|-----------------|--------------------|------------------|
| A | 1 | 1 | 1 |
| B | None | 1 | 1 |
| C | None | 1 | 1 |

5.1.4 Requirements and suggestions for installing Nebula Graph in production environments

Hardware requirements for production environments

| Item | Requirement |
|--------------------|----------------------|
| CPU architecture | x86_64 |
| Number of CPU core | 48 |
| Memory | 96 GB |
| Disk | 2 * 900 GB, NVMe SSD |

Supported operating systems for production environments

For now, we can only install Nebula Graph in the Linux system. To install Nebula Graph in a production environment, we recommend that you use any Linux system with kernel version 3.9 or above.

Users can adjust some of the kernel parameters to better accommodate the need for running Nebula Graph. For more information, see [kernel configuration](#).

Suggested service architecture for production environments

Danger

DO NOT deploy a cluster across IDCs.

| Process | Suggested number |
|---|------------------|
| metad (the metadata service process) | 3 |
| storaged (the storage service process) | 3 or more |
| graphd (the query engine service process) | 3 or more |

Each metad process automatically creates and maintains a replica of the metadata. Usually, you need to deploy three metad processes and only three.

The number of storaged processes does not affect the number of graph space replicas.

Users can deploy multiple processes on a single machine. For example, on a cluster of 5 machines (named as A, B, C, D, and E), you can deploy Nebula Graph as follows:

| Machine name | Number of metad | Number of storaged | Number of graphd |
|--------------|-----------------|--------------------|------------------|
| A | 1 | 1 | 1 |
| B | 1 | 1 | 1 |
| C | 1 | 1 | 1 |
| D | None | 1 | 1 |
| E | None | 1 | 1 |

5.1.5 Capacity requirements for running a Nebula Graph cluster

Users can estimate the memory, disk space, and partition number needed for a Nebula Graph cluster of 3 replicas as follows.

| Resource | Unit | How to estimate | Description |
|--|-------|---|---|
| Disk space for a cluster | Bytes | <code>the_sum_of_edge_number_and_vertex_number * average_bytes_of_properties * 6 * 120%</code> | - |
| Memory for a cluster | Bytes | <code>[the_sum_of_edge_number_and_vertex_number * 15 + the_number_of_RocksDB_instances * (write_buffer_size * max_write_buffer_number + rocksdb_block_cache)] * 120%</code> | <code>write_buffer_size</code> and <code>max_write_buffer_number</code> are RocksDB parameters. For more information, see MemTable . For details about <code>rocksdb_block_cache</code> , see Memory usage in RocksDB . |
| Number of partitions for a graph space | - | <code>the_number_of_disks_in_the_cluster * disk_partition_num_multiplier</code> | <code>disk_partition_num_multiplier</code> is an integer between 2 and 10 (both including). Its value depends on the disk performance. Use 2 for HDD. |

- Question 1: Why do we multiply the disk space and memory by 120%?

Answer: The extra 20% is for buffer.

- Question 2: How to get the number of RocksDB instances?

Answer: Each directory in the `--data_path` item in the `etc/nebula-storaged.conf` file corresponds to a RocksDB instance. Count the number of directories to get the RocksDB instance number.

Note

Users can decrease the memory size occupied by the bloom filter by adding `--enable_partitioned_index_filter=true` in `etc/nebula-storaged.conf`. But it may decrease the read performance in some random-seek cases.

5.1.6 FAQ

About storage devices

Nebula Graph is designed and implemented for NVMe SSD. All default parameters are optimized for the SSD devices and require extremely high IOPS and low latency.

- Due to the poor IOPS capability and long random seek latency, HDD is not recommended. Users may encounter many problems when using HDD.
- Do not use remote storage devices, such as NAS or SAN. Do not connect an external virtual hard disk based on HDFS or Ceph.
- Do not use RAID.
- Use local SSD devices.

About CPU architecture

Enterpriseonly

Nebula Graph 2.5.0 does not support running or compiling directly on the ARM architecture (including Apple Mac M1 or Huawei Kunpeng).

Last update: August 25, 2021

5.2 Compile and install Nebula Graph

5.2.1 Install Nebula Graph by compiling the source code

Installing Nebula Graph from the source code allows you to customize the compiling and installation settings and test the latest features.

Prerequisites

- Users have to prepare correct resources described in [Prepare resources for compiling, installing, and running Nebula Graph](#).
- The host to be installed with Nebula Graph has access to the Internet.

Installation steps

1. Use Git to clone the source code of Nebula Graph to the host.

- [Recommended] To install Nebula Graph v2.5.0, run the following command.

```
$ git clone --branch v2.5.0 https://github.com/vesoft-inc/nebula-graph.git
```

- To install the latest developing release, run the following command to clone the source code from the master branch.

```
$ git clone https://github.com/vesoft-inc/nebula-graph.git
```

2. Make the `nebula-graph` directory the current working directory.

```
$ cd nebula-graph
```

3. Create a `build` directory and make it the current working directory.

```
$ mkdir build && cd build
```

4. Generate Makefile with CMake.

Note

The installation path is `/usr/local/nebula` by default. To customize it, add the `-DCMAKE_INSTALL_PREFIX=<installation_path>` CMake variable in the following command.

For more information about CMake variables, see [CMake variables](#).

- If the source code of Nebula Graph v2.5.0 is installed and cloned in step 1, run the following command. The `-DNEBULA_COMMON_REPO_TAG` and `-DNEBULA_STORAGE_REPO_TAG` options are used to specify the correct branches of `nebula-common` and `nebula-storage` repositories to keep the releases of the Nebula Graph components consistent.

```
$ cmake -DENABLE_BUILD_STORAGE=on -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=Release \
-DNEBULA_COMMON_REPO_TAG=v2.5.0 -DNEBULA_STORAGE_REPO_TAG=v2.5.0 ..
```

- If the source code of the `master` branch is installed in step 1, run the following command.

```
$ cmake -DENABLE_BUILD_STORAGE=on -DENABLE_TESTING=OFF -DENABLE_MODULE_UPDATE=ON -DCMAKE_BUILD_TYPE=Release ..
```

5. Compile Nebula Graph.

Note

Check [Prepare resources for compiling, installing, and running Nebula Graph](#).

To speed up the compiling, use the `-j` option to set a concurrent number `N`. It should be $\lfloor \min(\text{CPU} \text{ core number}, \frac{\text{the_memory_size(GB)}}{2}) \rfloor$.

```
$ make -j{N} # E.g., make -j2
```

6. Install Nebula Graph.

```
$ sudo make install-all
```

7. The configuration files in the `etc/` directory (`/usr/local/nebula/etc` by default) are references. Users can create their own configuration files accordingly. If you want to use the scripts in the `script` directory to start, stop, restart, and kill the service, and check the service status, the configuration files have to be named as `nebula-graph.conf`, `nebula-metad.conf`, and `nebula-storaged.conf`.

Update the master branch

The source code of the master branch changes frequently. If the corresponding Nebula Graph release is installed, update it in the following steps.

1. In the `'nebula-graph/'` directory, run `'git pull upstream master'` to update the source code.
2. In the `'nebula-graph/modules/common/'` and `'nebula-graph/modules/storage/'` directories, run `'git pull upstream master'` separately.
3. In the `'nebula-graph/build/'` directory, run `'make -j{N}'` and `'make install-all'` again.

Next

- [Manage Nebula Graph services](#)
- [Connect to Nebula Graph](#)
- [Nebula Graph CRUD](#)

CMake variables

USAGE OF CMAKE VARIABLES

```
$ cmake -D<variable>=<value> ...
```

The following CMake variables can be used at the configure (cmake) stage to adjust the compiling settings.

ENABLE_BUILD_STORAGE

Starting from 2.5.0, Nebula Graph uses two separated github repositories of graph and storage for separated compiling. The `ENABLE_BUILD_STORAGE` variable is set to `OFF` by default so that the storage service is not installed together with the graph service.

If you are deploying Nebula Graph on a single host for testing, you can set `ENABLE_BUILD_STORAGE` to `ON` to download and install the storage service automatically.

CMAKE_INSTALL_PREFIX

`CMAKE_INSTALL_PREFIX` specifies the path where the service modules, scripts, configuration files are installed. The default path is `/usr/local/nebula`.

ENABLE_WERROR

`ENABLE_WERROR` is `ON` by default and it makes all warnings into errors. You can set it to `OFF` if needed.

ENABLE_TESTING

`ENABLE_TESTING` is `ON` by default and unit tests are built with the Nebula Graph services. If you just need the service modules, set it to `OFF`.

ENABLE_ASAN

`ENABLE_ASAN` is `OFF` by default and the building of ASan (AddressSanitizer), a memory error detector, is disabled. To enable it, set `ENABLE_ASAN` to `ON`. This variable is intended for Nebula Graph developers.

CMAKE_BUILD_TYPE

Nebula Graph supports the following building types of `MAKE_BUILD_TYPE`:

- `Debug`

The default value of `CMAKE_BUILD_TYPE`. It indicates building Nebula Graph with the debug info but not the optimization options.

- `Release`

It indicates building Nebula Graph with the optimization options but not the debug info.

- `RelWithDebInfo`

It indicates building Nebula Graph with the optimization options and the debug info.

- `MinSizeRel`

It indicates building Nebula Graph with the optimization options for controlling the code size but not the debug info.

CMAKE_C_COMPILER/CMAKE_CXX_COMPILER

Usually, CMake locates and uses a C/C++ compiler installed in the host automatically. But if your compiler is not installed at the standard path, or if you want to use a different one, run the command as follows to specify the installation path of the target compiler:

```
$ 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

`ENABLE_CCACHE` is `ON` by default and ccache (compiler cache) is used to speed up the compiling of Nebula Graph.

To disable `ccache`, setting `ENABLE_CCACHE` to `OFF` is not enough. On some platforms, the `ccache` installation hooks up or precedes the compiler. In such a case, you have to set an environment variable `export CCACHE_DISABLE=true` or add a line `disable=true` in `~/.ccache/ccache.conf` as well. For more information, see the [ccache official documentation](#).

NEBULA_THIRDPARTY_ROOT

`NEBULA_THIRDPARTY_ROOT` specifies the path where the third party software is installed. By default it is `/opt/vesoft/third-party`.

Examine problems

If the compiling fails, we suggest you:

1. Check whether the operating system release meets the requirements and whether the memory and hard disk space are sufficient.
2. Check whether the [third-party](#) is installed correctly.
3. Use `make -j1` to reduce the compiling concurrency.
4. Update the code `git pull`, compile again, and add the CMake option `-DENABLE_MODULE_UPDATE=ON` in step 4.

Last update: August 25, 2021

5.2.2 Install Nebula Graph with RPM or DEB package

RPM and DEB are common package formats on Linux systems. This topic shows how to quickly install Nebula Graph with the RPM or DEB package.

Note

For ways to deploy the Nebula Graph cluster, see [Deploy Nebula Graph cluster with RPM/DEB package](#).

Prerequisites

You should install wget before installing Nebula Graph with RPM/DEB package.

Download the package

DOWNLOAD THE PACKAGE FROM CLOUD SERVICE

- Download the released version.

URL:

```
//Centos 7
https://oss-cdn.nebula-graph.io/package/<release_version>/nebula-graph-<release_version>.el7.x86_64.rpm

//Centos 8
https://oss-cdn.nebula-graph.io/package/<release_version>/nebula-graph-<release_version>.el8.x86_64.rpm

//Ubuntu 1604
https://oss-cdn.nebula-graph.io/package/<release_version>/nebula-graph-<release_version>.ubuntu1604.amd64.deb

//Ubuntu 1804
https://oss-cdn.nebula-graph.io/package/<release_version>/nebula-graph-<release_version>.ubuntu1804.amd64.deb

//Ubuntu 2004
https://oss-cdn.nebula-graph.io/package/<release_version>/nebula-graph-<release_version>.ubuntu2004.amd64.deb
```

For example, download release package 2.5.0 for Centos 7.5:

```
 wget https://oss-cdn.nebula-graph.io/package/2.5.0/nebula-graph-2.5.0.el7.x86_64.rpm
 wget https://oss-cdn.nebula-graph.io/package/2.5.0/nebula-graph-2.5.0.el7.x86_64.rpm.sha256sum.txt
```

For example, download release package 2.5.0 for Ubuntu 1804:

```
wget https://oss-cdn.nebula-graph.io/package/2.5.0/nebula-graph-2.5.0.ubuntu1804.amd64.deb
wget https://oss-cdn.nebula-graph.io/package/2.5.0/nebula-graph-2.5.0.ubuntu1804.amd64.deb.sha256sum.txt
```

- Download the nightly version.

Danger

The nightly version is usually used to test new features. **DO NOT** use it in a production environment.

URL:

```
//Centos 7
https://oss-cdn.nebula-graph.io/package/v2-nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.el7.x86_64.rpm

//Centos 8
https://oss-cdn.nebula-graph.io/package/v2-nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.el8.x86_64.rpm

//Ubuntu 1604
https://oss-cdn.nebula-graph.io/package/v2-nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu1604.amd64.deb

//Ubuntu 1804
https://oss-cdn.nebula-graph.io/package/v2-nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu1804.amd64.deb

//Ubuntu 2004
https://oss-cdn.nebula-graph.io/package/v2-nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu2004.amd64.deb
```

For example, download release package 2.x for Centos 7.5 in 2021.08.18 :

```
wget https://oss-cdn.nebula-graph.io/package/v2-nightly/2021.08.18/nebula-graph-2021.08.18-nightly.el7.x86_64.rpm
wget https://oss-cdn.nebula-graph.io/package/v2-nightly/2021.08.18/nebula-graph-2021.08.18-nightly.el7.x86_64.rpm.sha256sum.txt
```

For example, download release package 2.x for Ubuntu 1804 in 2021.08.18 :

```
wget https://oss-cdn.nebula-graph.io/package/v2-nightly/2021.08.18/nebula8graph-2021.08.18-nightly.ubuntu1804.amd64.deb
wget https://oss-cdn.nebula-graph.io/package/v2-nightly/2021.08.18/nebula-graph-2021.08.18-nightly.ubuntu1804.amd64.deb.sha256sum.txt
```

Install Nebula Graph

- Use the following syntax to install with an RPM package.

```
$ sudo rpm -ivh --prefix=<installation_path> <package_name>
```

- Use the following syntax to install with a DEB package.

```
$ sudo dpkg -i --instdir=<installation_path> <package_name>
```

Note

The default installation path is `/usr/local/nebula/`.

Next

- [Manage Nebula Graph](#)
- [Connect to Nebula Graph](#)

Last update: August 24, 2021

5.3 Deploy Nebula Graph cluster

This topic describes how to manually deploy a Nebula Graph cluster.

Note

For now, Nebula Graph does not have an official deployment tool.

5.3.1 Prerequisites

[Prepare hardware](#) for deploying the cluster.

5.3.2 Step 1: Install Nebula Graph

Install Nebula Graph on each machine in the cluster. Available approaches of installation are as follows.

- [Install Nebula Graph with RPM or DEB package](#)
- [Install Nebula Graph by compiling the source code](#)

5.3.3 Step 2: Modify the configurations

To deploy Nebula Graph according to your requirements, you have to modify the configuration files. All the configuration files for Nebula Graph, including `nebula-graphd.conf`, `nebula-metad.conf`, and `nebula-storaged.conf`, are stored in the `etc` directory in the installation path.

You only need to modify the configuration for the corresponding service on the machines. For example, modify `nebula-graphd.conf` on the machines where you want to deploy the Graph Service.

For how to prepare the configuration files, see:

- [Meta Service configurations](#)
- [Graph Service configurations](#)
- [Storage Service configurations](#)

5.3.4 Step 3: Start the cluster

Start the corresponding service on each machine. The command to start the Nebula Graph services is as follows.

```
sudo /usr/local/nebula/scripts/nebula.service start <metad|graphd|storaged|all>
```

`/usr/local/nebula` is the default installation path for Nebula Graph. Use the actual path if you have customized the path.

For more information about how to start and stop the services, see [Manage Nebula Graph services](#).

5.3.5 Connect to the cluster

Connect to the Graph Service with a Nebula Graph client, such as Nebula Console. For more information, see [Connect to Nebula Graph](#).

5.3.6 Check the cluster status

After connecting to the Nebula Graph cluster, run `SHOW HOSTS` to check the cluster status.

Last update: April 22, 2021

5.4 Upgrade Nebula Graph to v2.0.0

This topic describes how to upgrade Nebula Graph to v2.0.0.

5.4.1 Limitations

- Rolling Upgrade is not supported. You must stop the Nebula Graph services before the upgrade.
- There is no upgrade script. You have to manually upgrade each server in the cluster.
- Supported versions:
 - From Nebula Graph [v1.2.0](#) to [Nebula Graph v2.0.0](#).
 - From Nebula Graph [v2.0.0-RC1](#) to Nebula Graph 2.0.0.
- This topic does not apply to scenarios where Nebula Graph is deployed with Docker, including Docker Swarm, Docker Compose, and Kubernetes.
- You must upgrade the old Nebula Graph services on the same machines they are deployed. **DO NOT** change the IP addresses, configuration files of the machines, and **DO NOT** change the cluster topology.
- The hard disk space of each machine should be three times as much as the space taken by the original data directories.
- Known issues that could cause data loss are listed on [GitHub known issues](#). The issues are all related to altering schema or default values.
- To connect to Nebula Graph 2.0.0, you must upgrade all the Nebula Graph clients. The communication protocols of the old versions and the latest versions are not compatible.
- The upgrade takes about 30 minutes in [this test environment](#).
- **DO NOT** use soft links to switch the data directories.
- You must have the sudo privileges to complete the steps in this topic.

5.4.2 Installation paths

Old installation path

By default, old versions of Nebula Graph are installed in `/usr/local/nebula/`, hereinafter referred to as `${nebula-old}` . The default configuration file path is `${nebula-old}/etc/` .

The data of the old Nebula Graph are stored by the Storage Service and the Meta Service. You can find the data paths as follows.

- Storage data path is defined by the `--data_path` option in the `${nebula-old}/etc/nebula-storaged.conf` file. The default path is `data/storage` .
- Meta data path is defined by the `--data_path` option in the `${nebula-old}/etc/nebula-metad.conf` file. The default path is `data/meta` .

Note

The actual paths in your environment may be different from those described in this topic. You can run the Linux command `ps -ef | grep nebula` to locate them.

New installation path

`${nebula-new}` represents the installation path of the new Nebula Graph version. An example for `${nebula-new}` is `/usr/local/nebula-new/` .

5.4.3 Steps

1. Stop all client connections. You can run the following commands on each Graph server to turn off the Graph Service and avoid dirty write.

```
> ${nebula-old}/scripts/nebula.service stop graphd
[INFO] Stopping nebula-graphd...
[INFO] Done
```

2. Run the following commands to stop all services of the old version Nebula Graph.

```
> ${nebula-old}/scripts/nebula.service stop all
[INFO] Stopping nebula-metad...
[INFO] Done
[INFO] Stopping nebula-graphd...
[INFO] Done
[INFO] Stopping nebula-storaged...
[INFO] Done
```

The Storage Service needs about 1 minute to flush data. Wait 1 minute and then run `ps -ef | grep nebula` to check and make sure that all the Nebula Graph services are stopped.

Note

If the services are not fully stopped in 20 minutes, stop upgrading and go to the [Nebula Graph community](#) for help.

3. Install the new version of Nebula Graph on each machine.

- To install with RPM/DEB packages, run the following command. For detailed steps, see [Install Nebula Graph with RPM or DEB package](#).

```
> sudo rpm --force -i --prefix=${nebula-new} ${nebula-package-name.rpm} # for CentOS/RedHat
> sudo dpkg -i --instdir==${nebula-new} ${nebula-package-name.deb} # for Ubuntu
```

- To install with the source code, follow the substeps. For detailed steps, see [Install Nebula Graph by compiling the source code](#)

1. Clone the source code.

```
> git clone --branch v2.0.0 https://github.com/vesoft-inc/nebula-graph.git
```

2. Configure CMake.

```
> cmake -DCMAKE_INSTALL_PREFIX=${nebula-new} -DENABLE_BUILD_STORAGE=on -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=Release -
-DNEBULA_COMMON_REPO_TAG=v2.0.0 -DNEBULA_STORAGE_REPO_TAG=v2.0.0 ..
```

4. Copy the configuration files from the old path to the new path.

```
> cp -rf ${nebula-old}/etc ${nebula-new}/
```

5. Follow the substeps to prepare the Meta servers (usually 3 of them in a cluster).

Note

You must make sure that this step is applied on every Meta server.

- a. Locate the old Meta [data path](#) and copy the data files to the new path.

```
> mkdir -p ${nebula-new}/data/meta/
> cp -r ${nebula-old}/data/meta/* ${nebula-new}/data/meta/
```

- b. Modify the new Meta configuration files:

```
> vim ${nebula-new}/nebula-metad.conf
```

[Optional] Add the following parameters in the Meta configuration files if you need them.

- `--null_type=false` : Disables the support for using `NULL` as schema properties after the upgrade. The default value is `true`. When set to `false`, you must specify a [default value](#) when altering tags or edge types, otherwise, data reading fails.
- `--string_index_limit=32` : Specifies the index length for string values as 32. The default length is 64.

6. Prepare the Storage configuration files on each Storage server.

- If the old Storage data path is not the default setting `--data_path=data/storage`, Modify the Storage configuration file and change the value of `--data_path` as the new data path.

```
> vim ${nebula-new}/nebula-storaged.conf
```

- Create the new Storage data directories.

```
> mkdir -p ${nebula-new}/data/storage/
```

Note

If the `--data_path` default value has been modified, create the Storage data directories according to the modification.

7. Start the new Meta Service.

- Run the following command on each Meta server.

```
$ sudo ${nebula-new}/scripts/nebula.service start metad
[INFO] Starting nebula-metad...
[INFO] Done
```

- Check if every `nebula-metad` process is started normally.

```
$ ps -ef |grep nebula-metad
```

- Check if there is any error information in the Meta logs in `${nebula-new}/logs`. If any `nebula-metad` process cannot start normally, stop upgrading, start the Nebula Graph services from the old directories, and take the error logs to the [Nebula Graph community](#) for help.

8. Run the following commands to upgrade the Storage data format.

```
$ sudo ${nebula-new}/bin/db_upgrader \
--src_db_path=<old_storage_directory_path> \
--dst_db_path=<new_storage_directory_path> \
--upgrade_meta_server=<meta_server_ip1>:<port1>[,<meta_server_ip2>:<port2>,...] \
--upgrade_version=<old_nebula_version> \
```

The parameters are described as follows.

- `--src_db_path` : Specifies the absolute path of the **OLD** Storage data directories. Separate multiple paths with commas, without spaces.
- `--dst_db_path` : Specifies the absolute path of the **NEW** Storage data directories. Separate multiple paths with commas, without spaces. The paths must correspond to the paths set in `--src_db_path` one by one.

Danger

Don't mix up the preceding two parameters, otherwise, the old data will be damaged during the upgrade.

- `--upgrade_meta_server` : Specifies the addresses of the new Meta servers that you started in step 7.
- `--upgrade_version` : If the old Nebula Graph version is v1.2.0, set the parameter value to `1`. If the old version is v2.0.0-RC1, set the value to `2`.

Danger

Don't set the value to other numbers.

Example of upgrading from v1.2.0:

```
$ sudo /usr/local/nebula_new/bin/db_upgrader \
--src_db_path=/usr/local/nebula/data/storage/data1/,/usr/local/nebula/data/storage/data2/ \
--dst_db_path=/usr/local/nebula_new/data/storage/data1/,/usr/local/nebula_new/data/storage/data2/ \
--upgrade_meta_server=192.168.8.14:45500,192.168.8.15:45500,192.168.8.16:45500 \
--upgrade_version=1
```

Example of upgrading from v2.0.0-RC1:

```
$ sudo /usr/local/nebula_new/bin/db_upgrader \
--src_db_path=/usr/local/nebula/data/storage/ \
--dst_db_path=/usr/local/nebula_new/data/storage/ \
--upgrade_meta_server=192.168.8.14:9559,192.168.8.15:9559,192.168.8.16:9559 \
--upgrade_version=2
```

Note

Make sure that all the Storage servers have finished the upgrade. If anything goes wrong:

- Stop upgrading.
- Stop all the Meta servers.
- Start the Nebula Graph services from the old directories.
- Go to the [Nebula Graph community](#) for help.

9. Start the new Storage Service on each Storage server.

```
$ sudo ${nebula-new}/scripts/nebula.service start storaged
$ sudo ${nebula-new}/scripts/nebula.service status storaged
```

Note

If this step goes wrong on any server:

- Stop upgrading.
- Stop all the Meta servers and Storage servers.
- Start the Nebula Graph services from the old directories.
- Take the logs in `${nebula-new}/logs/` to the [Nebula Graph community](#) for help.

10. Start the new Graph Service on each Graph server.

```
$ sudo ${nebula-new}/scripts/nebula.service start graphd
$ sudo ${nebula-new}/scripts/nebula.service status graphd
```

Note

If this step goes wrong on any server:

- Stop upgrading.
- Stop all the Meta servers, Storage servers, and Graph servers.
- Start the Nebula Graph services from the old directories.
- Take the logs in `${nebula-new}/logs/` to the [Nebula Graph community](#) for help.

11. Connect to Nebula Graph with the new version (v2.0.0 or later) of [Nebula Console](#). Verify if the Nebula Graph services are available and if the data can be accessed normally.

The command for connection, including the IP address and port of the Graph Service, is the same as the old one.

The following statements may help in this step.

```
nebula> SHOW HOSTS;
nebula> SHOW SPACES;
nebula> USE <space_name>
nebula> SHOW PARTS;
nebula> SUBMIT JOB STATS;
nebula> SHOW STATS;
```

Danger

Don't use Nebula Console versions prior to v2.0.0.

12. Upgrade other Nebula Graph clients.

You must upgrade all other clients to corresponding v2.0.0 versions. The clients include but are not limited to the following ones. Find the v2.0.0 branch for each client.

- [studio](#)
- [python](#)
- [java](#)
- [go](#)
- [c++](#)
- [flink-connector](#)
- [spark-util](#)
- [benchmark](#)

Note

- Communication protocols of the v2.0.0 versions are not compatible with that of the historical versions. To upgrade the clients, you must compile the v2.0.0 source code of the clients or download corresponding binaries.
- Tip for maintenance: The data path after the upgrade is `${nebula-new} / .` Modify relative paths for hard disk monitor systems or log ELK.

5.4.4 Upgrade failure and rollback

If the upgrade fails, stop all Nebula Graph services of the new version, and start the services of the old version.

All Nebula Graph clients in use must be switched to the old version.

5.4.5 Appendix 1: Test Environment

The test environment for this topic is as follows:

- Machine specifications: 32 CPU cores, 62 GB memory, and SSD.
- Data size: 100 GB of Nebula Graph 1.2.0 LDBC test data, with 1 graph space, 24 partitions, and 92 GB of data directory size.
- Concurrent configuration:

| Parameter | Default value | Applied value in the Tests |
|-------------------------------------|---------------|----------------------------|
| <code>--max_concurrent</code> | 5 | 5 |
| <code>--max_concurrent_parts</code> | 10 | 24 |
| <code>--write_batch_num</code> | 100 | 100 |

The upgrade cost 21 minutes in all, including 21 minutes of compaction.

5.4.6 Appendix 2: Nebula Graph V2.0.0 code address and commit ID

| Code address | Commit ID |
|---------------------------|-----------|
| Graph Service | 7923a45 |
| Storage and Meta Services | 761f22b |
| Common | b2512aa |

5.4.7 FAQ

Can I write through the client during the upgrade?

A: No. The state of the data written during this process is undefined.

Can I upgrade other old versions except for v1.2.0 or v2.0.0-RC1 to v2.0.0?

A: Upgrading from other old versions is not tested. Theoretically, versions between v1.0.0 and v1.2.0 could adopt the upgrade approach for v1.2.0. V2.x nightly versions cannot apply the solutions in this topic.

How to upgrade clients after the server upgrade?

A: See step 12 in this topic.

How to upgrade if a machine has only the Graph Service, but not the Storage Service?

A: Upgrade the Graph Service with the corresponding binary or rpm package.

How to resolve the error Permission denied?

A: Try again with the sudo privileges.

Is there any change in gflags?

A: Yes. For more information, see [known gflags changes](#).

What are the differences between deleting data then installing the new version and upgrading according to this topic?

A: The default configurations for v2.x and v1.x are different, including the ports used. The upgrade solution keeps the old configurations, and the delete-and-install solution uses the new configurations.

Is there a tool or solution for verifying data consistency after the upgrade?

A: No.

.....

Last update: May 12, 2021

5.5 Uninstall Nebula Graph

This topic describes how to uninstall Nebula Graph.

⚠ Caution

Before re-installing Nebula Graph on a machine, follow this topic to completely uninstall the old Nebula Graph, in case the remaining data interferes with the new services.

5.5.1 Prerequisites

You have stopped the Nebula Graph services. For more information, see [Manage Nebula Graph services](#).

5.5.2 Step 1: Delete data files of the Storage and Meta Services

If you have modified the `data_path` in the configuration files for the Meta Service and Storage Service, the directories where Nebula Graph stores data may not be in the installation path of Nebula Graph. Check the configuration files to confirm the data paths, and then manually delete the directories to clear all data.

For a Nebula Graph cluster, delete the data files of all Storage and Meta servers.

⚠ Caution

Make sure that you have backed up all important data.

1. Find the data paths in the [Storage Service disk settings](#) and [Meta Service storage settings](#). For example:

```
##### Disk #####
# Root data path. Split by comma. e.g. --data_path=/disk1/path1/,/disk2/path2/
# One path per Rocksdb instance.
--data_path=nebula/data/storage
```

2. Delete the directories found in step 1.

5.5.3 Step 2: Delete the installation directories

The default installation path is `/usr/local/nebula`. It can be modified while installing Nebula Graph. Delete all installation directories, including the `cluster.id` files in them.

Uninstall Nebula Graph deployed with source code

Find the installation path of Nebula Graph, and delete the directories.

Uninstall Nebula Graph deployed with RPM packages

1. Run the following command to get the Nebula Graph version.

```
$ rpm -qa | grep "nebula"
```

The return message is as follows.

```
nebula-graph-2.5.0-1.x86_64
```

2. Run the following command to uninstall Nebula Graph.

```
sudo rpm -e <nebula_version>
```

For example:

```
sudo rpm -e nebula-graph-2.5.0-1.x86_64
```

3. Delete the installation directories.

Uninstall Nebula Graph deployed with DEB packages

1. Run the following command to get the Nebula Graph version.

```
$ dpkg -l | grep "nebula"
```

The return message is as follows.

```
ii  nebula-graph  2.5.0  amd64      Nebula Package built using CMake
```

2. Run the following command to uninstall Nebula Graph.

```
sudo dpkg -r <nebula_version>
```

For example:

```
sudo dpkg -r nebula-graph
```

3. Delete the installation directories.

Uninstall Nebula Graph deployed with Docker Compose

1. In the `nebula-docker-compose` directory, run the following command to stop the Nebula Graph services.

```
docker-compose down -v
```

2. Delete the `nebula-docker-compose` directory.

Last update: May 10, 2021

6. Configurations and logs

6.1 Configurations

6.1.1 Configurations

This document gives some introduction to configurations in Nebula Graph.

For the path and usage of local configuration files for Nebula Graph services, see:

- [Meta configuration](#)
- [Graph configuration](#)
- [Storage configuration](#)

Get configurations

Most configurations are gflags. You can get all the gflags and the explanations by the following command.

```
<binary> --help
```

For example:

```
$ ./nebula-metad --help
$ ./nebula-graphd --help
$ ./nebula-storaged --help
$ ./nebula-console --help
```

Besides, you can get the values of running flags by `curl`-ing from the services.

For example:

```
$ curl 127.0.0.1:19559/flags # From Meta
$ curl 127.0.0.1:19669/flags # From Graph
$ curl 127.0.0.1:19779/flags # From Storage
```

Modify configurations

We suggest that you change configurations from local configure files. To change configurations from local files, follow these steps:

1. Add `--local_config=true` to each configuration file. The configuration files are stored in `/usr/local/nebula/etc/` by default. If you have customized your Nebula Graph installation directory, the path to your configuration files is `$pwd/nebula/etc/`.
2. Save your modification to the files.
3. Restart the Nebula Graph services.

⚠ Caution

Remember to add `--local_config=true` to each configuration file.

To make your modifications take effect, restart all the Nebula Graph services.

Legacy version compatibility

The `curl` commands and parameters in Nebula Graph v2.x. are different from Nebula Graph v1.x. Those `curl` commands in v1.x are deprecated now.

Last update: April 22, 2021

6.1.2 Meta Service configuration

Nebula Graph provides two initial configuration files for the Meta Service: `nebula-metad.conf.default` and `nebula-metad.conf.production`. You can use them in different scenarios. The default file path is `/usr/local/nebula/etc/`.

How to use the configuration files

The Meta Service gets its configuration from the `nebula-metad.conf` file. You have to remove the suffix `.default` or `.production` from an initial configuration file for the Meta Service to apply the configuration defined in it.

If you have modified the configuration in the file and want new configuration to take effect, add `--local_conf=true` at the top of the file. Otherwise, Nebula Graph reads the cached configuration.

About parameter values

If a parameter is not set in the configuration file, Nebula Graph uses the default value.

Note

The default value of a parameter in Nebula Graph may be different from the predefined value in the `.default` and `.production` files.

The predefined parameters in `nebula-metad.conf.default` and `nebula-metad.conf.production` are different. And not all parameters are predefined. This topic uses the parameters in `nebula-metad.conf.default`.

Nebula Graph provides two initial configuration files for the Meta Service: `nebula-metad.conf.default` and `nebula-metad.conf.production`. You can use them in different scenarios. The default file path is `/usr/local/nebula/etc/`.

Basic configurations

| Name | Predefine Value | Descriptions |
|----------------------------|------------------------------------|--|
| <code>daemonize</code> | <code>true</code> | When set to <code>true</code> , the process is a daemon process. |
| <code>pid_file</code> | <code>pids/nebula-metad.pid</code> | File to host the process ID. |
| <code>timezone_name</code> | - | Specifies the Nebula Graph time zone. This parameter is not predefined in the initial configuration files. You can manually set it if you need it. The system default value is <code>UTC+00:00:00</code> . For the format of the parameter value, see Specifying the Time Zone with TZ . For example, <code>--timezone_name=CST-8</code> represents the GMT+8 time zone. |

Note

- While inserting time-type property values except timestamps, Nebula Graph transforms them to a UTC time according to the time zone specified with the `timezone_name` parameter in the [configuration files](#). The time-type values returned by nGQL queries are all UTC time.
- `timezone_name` is only used to transform the data stored in Nebula Graph. Other time-related data of the Nebula Graph processes still uses the default time zone of the host, such as the log printing time.

Logging configurations

| Name | Predefine Value | Descriptions |
|-----------------|------------------|--|
| log_dir | logs | Directory to the Meta Service log. We recommend that you put logs on a different hard disk from the <code>data_path</code> . |
| minloglevel | 0 | Specifies the minimum log level. Available values are <code>0</code> (INFO), <code>1</code> (WARNING), <code>2</code> (ERROR), and <code>3</code> (FATAL). We suggest that you set <code>minloglevel</code> to <code>0</code> for debugging and <code>1</code> for production. When you set it to <code>4</code> , Nebula Graph does not print any logs. |
| v | 0 | Specifies the verbose log level. Available values are 0-4. The larger the value, the more verbose the log. |
| logbufsecs | 0 | Specifies the maximum time to buffer the logs. The configuration is measured in seconds. |
| stdout_log_file | metad-stdout.log | Specifies the filename for the stdout log. |
| stderr_log_file | metad-stderr.log | Specifies the filename for the stderr log. |
| stderrthreshold | 2 | Specifies the minimum level to copy the log messages to stderr. |

Networking configurations

| Name | Predefine Value | Descriptions |
|-------------------------|-----------------|---|
| meta_server_addrs | 127.0.0.1:9559 | Specifies the IP addresses and ports of all Meta Services. Separate multiple addresses with commas. |
| local_ip | 127.0.0.1 | Specifies the local IP for the Meta Service. |
| port | 9559 | Specifies RPC daemon listening port. The external port for the Meta Service is predefined to <code>9559</code> . The internal port is predefined to <code>port + 1</code> , i.e., <code>9560</code> . Nebula Graph uses the internal port for multi-replica interactions. |
| ws_ip | 0.0.0.0 | Specifies the IP address for the HTTP service. |
| ws_http_port | 19559 | Specifies the port for the HTTP service. |
| ws_h2_port | 19560 | Specifies the port for the HTTP2 service. |
| heartbeat_interval_secs | 10 | Specifies the default heartbeat interval in seconds. Make sure the <code>heartbeat_interval_secs</code> values for all services are the same, otherwise Nebula Graph CANNOT work normally. |

Note

We recommend that you use the real IP address in your configuration because sometimes `127.0.0.1` can not be parsed correctly.

Storage configurations

| Name | Predefine Value | Descriptions |
|-----------|---|--|
| data_path | data/meta (i.e. /usr/local/nebula/data/meta/) | Directory for cluster metadata persistence |

Misc configurations

| Name | Predefine Value | Descriptions |
|------------------------|------------------------|--|
| default_parts_num | 100 | Specifies the default partition number when you create a new graph space. |
| default_replica_factor | 1 | Specifies the default replica factor number when you create a new graph space. |

 RocksDB options

| Name | Predefine Value | Descriptions |
|------------------|------------------------|--|
| rocksdb_wal_sync | true | Enable or disable RocksDB WAL synchronization. Available values are <code>true</code> (enable) and <code>false</code> (disable). |

Last update: April 22, 2021

6.1.3 Graph Service configuration

Nebula Graph provides two initial configuration files for the Graph Service: `nebula-graphd.conf.default` and `nebula-graphd.conf.production`. You can use them in different scenarios. The default file path is `/usr/local/nebula/etc/`.

How to use the configuration files

The Graph Service gets its configuration from the `nebula-graphd.conf` file. You have to remove the suffix `.default` or `.production` from an initial configuration file for the Graph Service to apply the configuration defined in it.

If you have modified the configuration in the file and want new configuration to take effect, add `--local_conf=true` at the top of the file. Otherwise, Nebula Graph reads the cached configuration.

About parameter values

If a parameter is not set in the configuration file, Nebula Graph uses its default value.

Note

The default value of a parameter in Nebula Graph may be different from the predefined value in the `.default` and `.production` files.

The predefined parameters in `nebula-graphd.conf.default` and `nebula-graphd.conf.production` are different. And not all parameters are predefined. This topic uses the parameters in `nebula-graphd.conf.default`.

Basic configurations

| Name | Predefine Value | Descriptions |
|-------------------------------|-------------------------------------|--|
| <code>daemonize</code> | <code>true</code> | When set to <code>true</code> , the process is a daemon process. |
| <code>pid_file</code> | <code>pids/nebula-graphd.pid</code> | File to host the process ID. |
| <code>enable_optimizer</code> | <code>true</code> | When set to <code>true</code> , the optimizer is enabled. |
| <code>timezone_name</code> | - | Specifies the Nebula Graph time zone. This parameter is not predefined in the initial configuration files. You can manually set it if you need it. The system default value is <code>UTC+00:00:00</code> . For the format of the parameter value, see Specifying the Time Zone with TZ . For example, <code>--timezone_name=CST-8</code> represents the GMT+8 time zone. |

Note

- While inserting time-type property values except timestamps, Nebula Graph transforms them to a UTC time according to the time zone specified with the `timezone_name` parameter in the [configuration files](#). The time-type values returned by nGQL queries are all UTC time.
- `timezone_name` is only used to transform the data stored in Nebula Graph. Other time-related data of the Nebula Graph processes still uses the default time zone of the host, such as the log printing time.

Logging configurations

| Name | Predefine Value | Descriptions |
|-----------------|-------------------|--|
| log_dir | logs | Directory to the Graph Service log. We recommend that you put logs on a different hard disk from the <code>data_path</code> . |
| minloglevel | 0 | Specifies the minimum log level. Available values are <code>0</code> (INFO), <code>1</code> (WARNING), <code>2</code> (ERROR), and <code>3</code> (FATAL). We suggest that you set <code>minloglevel</code> to <code>0</code> for debugging and <code>1</code> for production. When you set it to <code>4</code> , Nebula Graph does not print any logs. |
| v | 0 | Specifies the verbose log level. Available values are 0-4. The larger the value, the more verbose the log. |
| logbufsecs | 0 | Specifies the maximum time to buffer the logs. The configuration is measured in seconds. |
| redirect_stdout | true | When set to <code>true</code> , <code>stdout</code> and <code>stderr</code> are redirected. |
| stdout_log_file | graphd-stdout.log | Specifies the filename for the <code>stdout</code> log. |
| stderr_log_file | graphd-stderr.log | Specifies the filename for the <code>stderr</code> log. |
| stderrthreshold | 2 | Specifies the minimum level to copy the log messages to <code>stderr</code> . |

Networking configurations

| Name | Predefine Value | Descriptions |
|---------------------------|-----------------|--|
| meta_server_addrs | 127.0.0.1:9559 | Specifies the IP addresses and ports of all Meta Services. Separate multiple addresses with commas. |
| local_ip | 127.0.0.1 | Specifies the local IP for the Graph Service. |
| listen_netdev | any | Specifies the network device to listen on. |
| port | 9669 | Specifies RPC daemon listening port. The external port for the Graph Service is 9669. |
| reuse_port | false | When set to <code>false</code> , the SO_REUSEPORT is closed. |
| listen_backlog | 1024 | Specifies the backlog for the listen socket. You must modify this configuration together with the <code>net.core.somaxconn</code> . |
| client_idle_timeout_secs | 0 | Specifies the time to close an idle connection. This configuration is measured in seconds. |
| session_idle_timeout_secs | 0 | Specifies the time to expire an idle session. This configuration is measured in seconds. |
| num_accept_threads | 1 | Specifies the thread number to accept incoming connections. |
| num_netio_threads | 0 | Specifies the networking IO threads number. 0 is the number of CPU cores. |
| num_worker_threads | 0 | Specifies the thread number to execute user queries. 0 is the number of CPU cores. |
| ws_ip | 0.0.0.0 | Specifies the IP address for the HTTP service. |
| ws_http_port | 19669 | Specifies the port for the HTTP service. |
| ws_h2_port | 19670 | Specifies the port for the HTTP2 service. |
| heartbeat_interval_secs | 10 | Specifies the default heartbeat interval in seconds. Make sure the <code>heartbeat_interval_secs</code> values for all services are the same, otherwise Nebula Graph CANNOT work normally. |
| storage_client_timeout_ms | - | Specifies the RPC connection timeout threshold between the Graph Service and the Storage Service. This parameter is not predefined in the initial configuration files. You can manually set it if you need it. The system default value is 60000 ms. |

Note

We recommend that you use the real IP address in your configuration because sometimes `127.0.0.1` can not be parsed correctly.

Charset and collate configurations

| Name | Predefine Value | Descriptions |
|-----------------|-----------------|--|
| default_charset | utf8 | Specifies the default charset when you create a new graph space. |
| default_collate | utf8_bin | Specifies the default collate when you create a new graph space. |

Authorization and authentication configurations

| Name | Predefine Value | Descriptions |
|------------------|-----------------|---|
| enable_authorize | false | When set to <code>false</code> , the system authentication is not enabled. For more information, see Authentication . |
| auth_type | password | Specifies the login method. Available values are <code>password</code> , <code>ldap</code> , and <code>cloud</code> . |

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=9669
```

If you have set `enable_authorize` to `false`, you can log in with any account and password. For example:

```
/usr/local/nebula/bin/nebula -u any -p 123 --addr=127.0.0.1 --port=9669
```

Last update: April 22, 2021

6.1.4 Storage Service configurations

Nebula Graph provides two initial configuration files for the Storage Service: `nebula-storaged.conf.default` and `nebula-storaged.conf.production`. You can use them in different scenarios. The default file path is `/usr/local/nebula/etc/`.

Note

Raft Listener is different from the Storage Service. For more information, see [Raft Listener](#).

How to use the configuration files

The Storage Service gets its configuration from the `nebula-storaged.conf` file. You have to remove the suffix `.default` or `.production` from an initial configuration file for the Storage Service to apply the configuration defined in it.

If you have modified the configuration in the file and want the new configuration to take effect, add `--local_conf=true` at the top of the file. Otherwise, Nebula Graph reads the cached configuration.

About parameter values

If a parameter is not set in the configuration file, Nebula Graph uses its default value.

Note

The default value of a parameter in Nebula Graph may be different from the predefined value in the `.default` and `.production` files.

The predefined parameter in `nebula-storaged.conf.default` and `nebula-storaged.conf.production` are different. And not all parameters are predefined. This topic uses the parameters in `nebula-storaged.conf.default`.

Basic configurations

| Name | Predefine Value | Descriptions |
|----------------------------|---------------------------------------|--|
| <code>daemonize</code> | <code>true</code> | When set to <code>true</code> , the process is a daemon process. |
| <code>pid_file</code> | <code>pids/nebula-storaged.pid</code> | File to host the process ID. |
| <code>timezone_name</code> | - | Specifies the Nebula Graph time zone. This parameter is not predefined in the initial configuration files. You can manually set it if you need it. The system default value is <code>UTC+00:00:00</code> . For the format of the parameter value, see Specifying the Time Zone with TZ . For example, <code>--timezone_name=CST-8</code> represents the GMT+8 time zone. |

Note

- While inserting time-type property values except timestamps, Nebula Graph transforms them to a UTC time according to the time zone specified with the `timezone_name` parameter in the [configuration files](#). The time-type values returned by nGQL queries are all UTC time.
- `timezone_name` is only used to transform the data stored in Nebula Graph. Other time-related data of the Nebula Graph processes still uses the default time zone of the host, such as the log printing time.

Logging configurations

| Name | Predefine Value | Descriptions |
|-----------------|--------------------|---|
| log_dir | logs | Directory to the Storage Service log. We recommend that you put logs on a different hard disk from the <code>data_path</code> . |
| minloglevel | 0 | Specifies the minimum log level. Available values are 0-3. 0, 1, 2, and 3 are <code>INFO</code> , <code>WARNING</code> , <code>ERROR</code> , and <code>FATAL</code> . We suggest that you set <code>minloglevel</code> to 0 for debug, 1 for production. When you set it to 4, Nebula Graph does not print any logs. |
| v | 0 | Specifies the verbose log level. Available values are 0-4. The larger the value, the more verbose the log. |
| logbufsecs | 0 | Specifies the maximum time to buffer the logs. The configuration is measured in seconds. |
| redirect_stdout | true | When set to <code>true</code> , <code>stdout</code> and <code>stderr</code> are redirected. |
| stdout_log_file | storage-stdout.log | Specifies the filename for the <code>stdout</code> log. |
| stderr_log_file | storage-stderr.log | Specifies the filename for the <code>stderr</code> log. |
| stderrthreshold | 2 | Specifies the minimum level to copy the log messages to <code>stderr</code> . Available values are 0-3. 0, 1, 2, and 3 are <code>INFO</code> , <code>WARNING</code> , <code>ERROR</code> , and <code>FATAL</code> . |

Networking configurations

| Name | Predefine Value | Descriptions |
|-------------------------|-----------------|---|
| meta_server_addrs | 127.0.0.1:9559 | Specifies the IP addresses and ports of all Meta Services. Separate multiple addresses with commas. |
| local_ip | 127.0.0.1 | Specifies the local IP for the Storage Service. |
| port | 9779 | Specifies RPC daemon listening port. The external port for Storage Service is predefined to 9779. The internal ports are predefined to <code>port -2</code> , <code>port -1</code> , and <code>port + 1</code> , i.e., 9777, 9778, and 9780. Nebula Graph uses the internal ports for multi-replica interactions. |
| ws_ip | 0.0.0.0 | Specifies the IP address for the HTTP service. |
| ws_http_port | 19779 | Specifies the port for the HTTP service. |
| ws_h2_port | 19780 | Specifies the port for the HTTP2 service. |
| heartbeat_interval_secs | 10 | Specifies the default heartbeat interval in seconds. Make sure the <code>heartbeat_interval_secs</code> values for all services are the same, otherwise Nebula Graph CANNOT work normally. |

Note

We recommend that you use the real IP address in your configuration because sometimes `127.0.0.1` can not be parsed correctly.

Raft configurations

| Name | Predefine Value | Descriptions |
|------------------------------|-----------------|--|
| raft_heartbeat_interval_secs | 30 | Specifies the timeout for the Raft election. The configuration is measured in seconds. |
| raft_rpc_timeout_ms | 500 | Specifies the timeout for the Raft RPC. The configuration is measured in milliseconds. |
| wal_ttl | 14400 | Specifies the recycle RAFT wal time. The configuration is measured in seconds. |

Disk configurations

| Name | Predefine Value | Descriptions |
|---------------------------------|--------------------------|--|
| data_path | data/storage | Specifies the root data path. Separate multiple paths with commas. |
| rocksdb_batch_size | 4096 | Specifies the block cache for a batch operation. The configuration is measured in bytes. |
| rocksdb_block_cache | 4 | Specifies the block cache for BlockBasedTable. The configuration is measured in megabytes. |
| engine_type | rocksdb | Specifies the engine type. |
| rocksdb_compression | lz4 | Specifies the compression algorithm for RocksDB. Available values are <code>lz4</code> , <code>lz4hc</code> , <code>zlib</code> , <code>bzip2</code> , and <code>zstd</code> . |
| rocksdb_compression_per_level | \ | Specifies compression for each level. |
| enable_rocksdb_statistics | false | When set to <code>false</code> , RocksDB statistics is disabled. |
| rocksdb_stats_level | kExceptHistogramOrTimers | Specifies the stats level for RocksDB. Available values are <code>kExceptHistogramOrTimers</code> , <code>kExceptTimers</code> , <code>kExceptDetailedTimers</code> , <code>kExceptTimeForMutex</code> , and <code>kAll</code> . |
| enable_rocksdb_prefix_filtering | false | When set to <code>true</code> , the prefix bloom filter for RocksDB is enabled. Enabled bloom filter reduces memory usage. |
| rocksdb_filtering_prefix_length | 12 | Specifies the prefix length for each key. Available values are <code>12</code> and <code>16</code> . |

RocksDB options

The format of the RocksDB options is `{"<option_name>":<option_value>"}`. Multiple options are separated with commas.

| Name | Predefine Value | Descriptions |
|-----------------------------------|--|--|
| rocksdb_db_options | {} | Specifies the RocksDB options. |
| rocksdb_column_family_options | {"write_buffer_size": "67108864", "max_write_buffer_number": "4", "max_bytes_for_level_base": "268435456"} | Specifies the RocksDB column family options. |
| rocksdb_block_based_table_options | {"block_size": "8192"} | Specifies the RocksDB block based table options. |

Available `rocksdb_db_options` and `rocksdb_column_family_options` are listed as follows.

- `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
```

- `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
```

For more information about RocksDB configuration, see [RocksDB official documentation](#)

For 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 affected compared to direct truncation due to the probability calculation.

Storage configuration for large dataset

When you have a large dataset (in the RocksDB directory) and your memory is tight, we suggest that you set the `enable_partitioned_index_filter` parameter to `true`. For example, 100 vertices + 100 edges require 300 key-values. Each key takes 10bit in memory. Then you can calculate your own memory usage.

Last update: May 10, 2021

6.1.5 Kernel configurations

This document gives some introductions to the Kernel configurations in Nebula Graph.

ulimit

ULIMIT -C

`ulimit -c` limits the size of the core dumps. We recommend that you set it to `unlimited`. The command is:

```
ulimit -c unlimited
```

ULIMIT -N

`ulimit -n` limits the number of open files. We recommend that you set it to more than 100,000. For example:

```
ulimit -n 130000
```

Memory

VM.SWAPPINESS

`vm.swappiness` is the percentage of the free memory before starting swap. The greater the value, the more likely the swap occurs. We recommend that you set it to 0. When set to 0, the page cache is removed first. Note that when `vm.swappiness` is 0, it does not mean that there is no swap.

VM.MIN_FREE_KBYTES

`vm.min_free_kbytes` is used to force the Linux VM to keep a minimum number of kilobytes free. If you have a large system memory, we recommend that you increase this value. For example, if your physical memory 128GB, set it to 5GB. If the value is not big enough, the system cannot apply for enough continuous physical memory.

VM.MAX_MAP_COUNT

`vm.max_map_count` limits the maximum number of vma (virtual memory area) for a process. The default value is `65530`. It is enough for most applications. If your memory application fails because the memory consumption is large, increase the `vm.max_map_count` value.

VM.OVERCOMMIT_MEMORY

`vm.overcommit_memory` contains a flag that enables memory overcommitment. We recommend that you set the default value 0 or 1. DO NOT set it to 2.

VM.DIRTY_*

These values control the aggressiveness of the dirty page cache for the system. For write-intensive scenarios, you can make adjustments based on your needs (throughput priority or delay priority). We recommend that you use the system default value.

TRANSPARENT HUGE PAGE

For better delay performance, you must delete the transparent huge pages (THP). 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

The default value for this parameter is `1`. If set, the congestion window is timed out after an idle period. We recommend that you set it to 0, especially for long fat links (high latency and large bandwidth).

NET.CORE.SOMAXCONN

`net.core.somaxconn` is the maximum number of the backlogged sockets. The default value is `128`. For scenarios with a large number of burst connections, we recommend that you set it to greater than `1024`.

NET.IPV4.TCP_MAX_SYN_BACKLOG

The maximum number of remembered connection requests. The setting rule for this parameter is the same as that of `net.core.somaxconn`.

NET.CORE.NETDEV_MAX_BACKLOG

It determines the maximum number of packets. We recommend that you increase it to greater than 10,000, especially for 10G network adapters. The default value is `1000`.

NET.IPV4.TCP_KEEPALIVE_*

Keep alive parameters for the TCP connections. For applications that use a 4-layer transparent load balancer, if the idle connection is disconnected unexpectedly, decrease `tcp_keepalive_time` and `tcp_keepalive_intvl`.

NET.IPV4.TCP_RMEM/WMEM

The minimum, default, and maximum size of the TCP socket receive buffer. For long fat links, we recommend that you increase the default value to `bandwidth * RTT`.

SCHEDULER

For SSD devices, we recommend that you set `/sys/block/DEV_NAME/queue/scheduler` to `noop` or `none`.

Other parameters**KERNEL.CORE_PATTERN**

we recommend that you 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 your modification takes effect immediately.
- `sysctl -p` loads parameter values from related configuration files.

INTRODUCTION TO ULIMIT

`ulimit` sets the resource threshold for the current shell session. Please note that:

- Changes made by the `ulimit` command are valid only for the current session (and child processes).
- `ulimit` cannot adjust the (soft) threshold of a resource to a value greater than the current hard value.
- Ordinary users cannot adjust the hard threshold (even by using `sudo`) through this command.
- To modify on the system level, or adjust the hard threshold, edit the `/etc/security/limits.conf` file. But this method needs to re-log in to take effect.

PRLIMIT

`prlimit` gets and sets process resource limits. You can modify the hard threshold by using it and the `sudo` command. Together with the `sudo` command, the hard threshold can be modified. For example, `prlimit --nofile = 130000 --pid = $$` adjusts the maximum number of open files permitted by the current process to `14000`. And the modification takes effect immediately. Note that this command is only available in RedHat 7u or later OS versions.

6.2 Log management

6.2.1 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. Not all flags are listed because changing some flags can be dangerous. Read the response explanation and the source code before you change these not documented parameters. To get all the available flags for a process, use this command:

```
> curl ${ws_ip}:${ws_port}/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 `19559`, `19779`, and `19669`, respectively.

Note

If you changed the runtime log level, then restart the services, the log level changes to the configuration file specifications. For more information, see [Storage Service configurations](#).

For example, check the `minloglevel` for the `storaged` service:

```
> curl 127.0.0.1:19559	flags | grep minloglevel
```

To change the log level for a process, use these commands. For example, you can change the log severity level the **the most detailed**.

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":0,"v":3}' "127.0.0.1:19779/flags" # storaged
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":0,"v":3}' "127.0.0.1:19669/flags" # graphd
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":0,"v":3}' "127.0.0.1:19559/flags" # metad
```

To change the severity of the storage log, replace the port in the preceding command with `storage` port.

Note

Nebula Graph only supports modifying the graph and storage log severity by using the 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 -X PUT -H "Content-Type: application/json" -d '{"minloglevel":3,"v":0}' "127.0.0.1:19779/flags" # storaged
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":3,"v":0}' "127.0.0.1:19669/flags" # graphd
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":3,"v":0}' "127.0.0.1:19559/flags" # metad
```

Last update: April 22, 2021

7. Monitor and metrics

7.1 Query Nebula Graph metrics

Nebula Graph supports querying the monitoring metrics through HTTP ports.

7.1.1 Metrics

Each metric of Nebula Graph consists of three fields: name, type, and time range. The fields are separated by periods, for example, `num_queries.sum.600`. The detailed description is as follows.

| Field | Example | Description |
|-------------|--------------------------|--|
| Metric name | <code>num_queries</code> | Indicates the function of the metric. |
| Metric type | <code>sum</code> | Indicates how the metrics are collected. Supported types are SUM, COUNT, AVG, RATE, and the P-th sample quantiles such as P75, P95, P99, and P99.9. |
| Time range | <code>600</code> | The time range in seconds for the metric collection. Supported values are 5, 60, 600, and 3600, representing the last 5 seconds, 1 minute, 10 minutes, and 1 hour. |

Different Nebula Graph services (Graph, Storage, or Meta) support different metrics, for more information, see Metric list (TODO: doc).

7.1.2 Query metrics over HTTP

Syntax

```
$ curl -G "http://<ip>:<port>/stats?stats=<metric_name_list>[&format=json]"
```

| Parameter | Description |
|-------------------------------|---|
| <code>ip</code> | The IP address of the server. You can find it in the configuration file in the installation directory. |
| <code>port</code> | The HTTP port of the server. You can find it in the configuration file in the installation directory. The default ports are 19559 (Meta), 19669 (Graph), and 19779 (Storage). |
| <code>metric_name_list</code> | The metrics names. Multiple metrics are separated by commas (,). |
| <code>&format=json</code> | Optional. Returns the result in the JSON format. |

Note

If Nebula Graph is deployed with [Docker Compose](#), run `docker-compose ps` to check the ports that are mapped from the service ports inside of the container and then query through them.

Example

- Query a single metric
- Query the query number in the last 10 minutes in the Graph Service.

```
$ curl -G "http://192.168.8.40:19669/stats?stats=num_queries.sum.600"
num_queries.sum.600=400
```

- Query multiple metrics

Query the following metrics together: * The average heartbeat latency in the last 1 minute. * The average latency of the slowest 1% heartbeats, i.e., the P99 heartbeats, in the last 10 minutes.

```
$ curl -G "http://192.168.8.40:19559/stats?stats=heartbeat_latency_us.avg.60,heartbeat_latency_us.p99.600"
heartbeat_latency_us.avg.60=281
heartbeat_latency_us.p99.600=985
```

- Return a JSON result.

Query the number of new vertices in the Storage Service in the last 10 minutes and return the result in the JSON format.

```
$ curl -G "http://192.168.8.40:19779/stats?stats=num_add_vertices.sum.600&format=json"
[{"value":1,"name":"num_add_vertices.sum.600"}]
```

- Query all metrics in a service.

If no metric is specified in the query, Nebula Graph returns all metrics in the service.

```
$ curl -G "http://192.168.8.40:19559/stats"
heartbeat_latency_us.avg.5=304
heartbeat_latency_us.avg.60=308
heartbeat_latency_us.avg.600=299
heartbeat_latency_us.avg.3600=285
heartbeat_latency_us.p75.5=652
heartbeat_latency_us.p75.60=669
heartbeat_latency_us.p75.600=651
heartbeat_latency_us.p75.3600=642
heartbeat_latency_us.p95.5=930
heartbeat_latency_us.p95.60=963
heartbeat_latency_us.p95.600=933
heartbeat_latency_us.p95.3600=929
heartbeat_latency_us.p99.5=986
heartbeat_latency_us.p99.60=1409
heartbeat_latency_us.p99.600=989
heartbeat_latency_us.p99.3600=986
num_heartbeats.rate.5=0
num_heartbeats.rate.60=0
num_heartbeats.rate.600=0
num_heartbeats.rate.3600=0
num_heartbeats.sum.5=2
num_heartbeats.sum.60=40
num_heartbeats.sum.600=394
num_heartbeats.sum.3600=2364
```

Last update: April 22, 2021

8. Data security

8.1 Authentication and authorization

8.1.1 Authentication

Nebula Graph replies on local authentication or LDAP authentication to implement access control.

Nebula Graph creates a session when a client connects to it. The session stores information about the connection, including the user information.

By default, authentication is disabled and Nebula Graph allows connections with any username and password. If the authentication system is enabled, Nebula Graph checks a session according to the authentication configuration, and decides whether the session should be allowed or denied.

Local authentication

Local authentication indicates that usernames and passwords are stored locally on the server, with the passwords encrypted.

ENABLE LOCAL AUTHENTICATION

1. In the `/usr/local/nebula/etc/nebula-graphd.conf` file, set `--enable_authorize=true` and save the modification.

Note

`/usr/local/nebula/` is the default installation path for Nebula Graph. If you have changed it, use the actual path.

2. Restart the Nebula Graph services. For how to restart, see [Manage Nebula Graph services](#).

Note

You can use the username `root` and password `nebula` to log into Nebula Graph after enabling local authentication. This account has the build-in God role. For more information about roles, see [Roles and privileges](#).

LDAP authentication

Lightweight Directory Access Protocol (LDAP), is a lightweight client-server protocol for accessing directories and building a centralized account management system.

LDAP authentication and local authentication can be enabled at the same time, but LDAP authentication has a higher priority. If the local authentication server and the LDAP server both have the information of user `Amber`, Nebula Graph reads from the LDAP server first.

ENABLE LDAP AUTHENTICATION

The Nebula Graph Enterprise Edition supports LDAP authentication. For how to enable LDAP, see [Authenticate with an LDAP server \(TODO: doc\)](#).

Last update: April 22, 2021

8.1.2 User management

This topic describes how to manage users and roles.

By default, Nebula Graph allows connections with any username and password. After [enabling authentication](#), only valid users can connect to Nebula Graph and access the resources according to the [user roles](#).

CREATE USER

The `root` user with the GOD role can run `CREATE USER` to create a new user.

- Syntax

```
CREATE USER [IF NOT EXISTS] <user_name> [WITH PASSWORD '<password>'];
```

- Example

```
nebula> CREATE USER user1 WITH PASSWORD 'nebula';
```

GRANT ROLE

Users with the GOD role or the ADMIN role can run `GRANT ROLE` to assign a built-in role in a graph space to a user. For more information about Nebula Graph built-in roles, see [Roles and privileges](#)

Note

If the target user is connected to Nebula Graph when running `GRANT ROLE`, the new role takes effect when the user logs out and logs in again.

- Syntax

```
GRANT ROLE <role_type> ON <space_name> TO <user_name>;
```

- Example

```
nebula> GRANT ROLE USER ON basketballplayer TO user1;
```

REVOKE ROLE

Users with the GOD role or the ADMIN role can run `REVOKE ROLE` to revoke a user's role in a graph space.

Note

If the target user is connected to Nebula Graph when running `REVOKE ROLE`, the old role still takes effect until the user logs out.

- Syntax

```
REVOKE ROLE <role_type> ON <space_name> FROM <user_name>;
```

- Example

```
nebula> REVOKE ROLE USER ON basketballplayer FROM user1;
```

CHANGE PASSWORD

With the correct username and password, users can run `CHANGE PASSWORD` to set a new password for a user.

- Syntax

```
CHANGE PASSWORD <user_name> FROM '<old_password>' TO '<new_password>';
```

- Example

```
nebula> CHANGE PASSWORD user1 FROM 'nebula' TO 'nebula123';
```

ALTER USER

The `root` user with the GOD role can run `ALTER USER` to set a new password for a user.

- Syntax

```
ALTER USER <user_name> WITH PASSWORD '<password>';
```

- Example

```
nebula> ALTER USER user1 WITH PASSWORD 'nebula';
```

DROP USER

The `root` user with the GOD role can run `DROP USER` to remove a user.

 **Note**

Removing a user does not close the user's current session, and the user role still takes effect in the session until the session is closed.

- Syntax

```
DROP USER [IF EXISTS] <user_name>;
```

- Example

```
nebula> DROP USER user1;
```

SHOW USERS

The `root` user with the GOD role can run `SHOW USERS` to list all the users.

- Syntax

```
SHOW USERS;
```

- Example

```
nebula> SHOW USERS;
+-----+
| Account |
+-----+
| "test1" |
+-----+
| "test2" |
+-----+
| "test3" |
+-----+
```

8.1.3 Roles and privileges

A role is a collection of privileges. You can assign a role to a user for access control.

Built-in roles

Nebula Graph does not support custom roles, but it has multiple built-in roles:

- GOD
 - GOD is the original role with all privileges not limited to graph spaces. It is similar to `root` in Linux and `administrator` in Windows.
 - When the Meta Service is initialized, the one and only GOD role user `root` is automatically created with the password `nebula`.
-  **Note**

Modify the password for `root` as soon as possible for security.

 - The default username `root` is immutable.
 - If [authentication](#) is disabled, you can use any username and password to connect to Nebula Graph. This user is regarded as the GOD role.
- ADMIN
 - An ADMIN role can read and write both the Schema and the data in a specific graph space.
 - An ADMIN role of a graph space can grant DBA, USER, and GUEST roles in the graph space to other users.
- DBA
 - A DBA role can read and write both the Schema and the data in a specific graph space.
 - A DBA role of a graph space CANNOT grant roles to other users.
- USER
 - A USER role can read and write data in a specific graph space.
 - The Schema information is read-only to the USER roles in a graph space.
- GUEST
 - A GUEST role can only read the Schema and the data in a specific graph space.

 **Note**

A user can have only one role in a graph space.

Role privileges and allowed nGQL

The privileges of roles and the nGQL statements that each role can use are listed as follows.

| Privilege | God | Admin | DBA | User | Guest | Allowed nGQL |
|-----------------|-----|-------|-----|------|-------|---|
| Read space | Y | Y | Y | Y | Y | USE, DESCRIBE SPACE |
| Write space | Y | | | | | CREATE SPACE, DROP SPACE, CREATE SNAPSHOT, DROP SNAPSHOT, BALANCE, ADMIN, CONFIG, INGEST, DOWNLOAD |
| Read schema | Y | Y | Y | Y | Y | DESCRIBE TAG, DESCRIBE EDGE, DESCRIBE TAG INDEX, DESCRIBE EDGE INDEX |
| Write schema | Y | Y | Y | | | CREATE TAG, ALTER TAG, CREATE EDGE, ALTER EDGE, DROP TAG, DROP EDGE, CREATE TAG INDEX, CREATE EDGE INDEX, DROP TAG INDEX, DROP EDGE INDEX |
| Write user | Y | | | | | CREATE USER, DROP USER, ALTER USER |
| Write role | Y | Y | | | | GRANT, REVOKE |
| Read data | Y | Y | Y | Y | Y | GO, SET, PIPE, MATCH, ASSIGNMENT, LOOKUP, YIELD, ORDER BY, FETCH VERTICES, Find, FETCH EDGES, FIND PATH, LIMIT, GROUP BY, RETURN |
| Write data | Y | Y | Y | Y | | BUILD TAG INDEX, BUILD EDGE INDEX, INSERT VERTEX, UPDATE VERTEX, INSERT EDGE, UPDATE EDGE, DELETE VERTEX, DELETE EDGES |
| Show operations | Y | Y | Y | Y | Y | SHOW, CHANGE PASSWORD |

Note

- The results of `SHOW` operations are limited to the role of a user. For example, all users can run `SHOW SPACES`, but the results only include the graph spaces that the users have privileges.
- Only the GOD role can run `SHOW USERS` and `SHOW SNAPSHTOTS`.

Last update: April 22, 2021

8.2 Backup and restore data with snapshots

Nebula Graph supports using snapshots to backup and restore data.

8.2.1 Authentication and snapshots

Nebula Graph [authentication](#) is disabled by default. In this case, All users can use the snapshot feature.

If authentication is enabled, only the GOD-role user can use the snapshot function. For more information about roles, see [Roles and privileges](#).

8.2.2 Precautions

- To prevent data loss, create a snapshot as soon as the system structure changes, for example, after operations such as `ADD HOST`, `DROP HOST`, `CREATE SPACE`, `DROP SPACE`, and `BALANCE` are performed.
- Nebula Graph cannot automatically delete the invalid files created by a failed snapshot task, you have to manually delete them by using `DROP SNAPSHOT`.
- Customizing the storage path for the snapshots is not supported for now.

8.2.3 Snapshot form and path

Nebula Graph snapshots are in the form of directories with names like `SNAPSHOT_2021_03_09_08_43_12`. The suffix `2021_03_09_08_43_12` is generated automatically based on the creation time.

When a snapshot is created, snapshot directories will be automatically created in the `checkpoints` directory on the leader Meta server and each Storage server.

To fast locate the path where the snapshots are stored, you can use the Linux command `find`. For example:

```
$ find |grep 'SNAPSHOT_2021_03_11_07_30_36'
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_11_07_30_36
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_11_07_30_36/data
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_11_07_30_36/data/000081.sst
...
```

Note

For how to get the snapshot name, see [View snapshots](#).

8.2.4 Create a snapshot

Run `CREATE SNAPSHOT` to create a snapshot for all the graph spaces based on the current time for Nebula Graph.

Note

Creating a snapshot for a specific graph space is not supported yet.

If the creation fails, [delete the snapshot](#) and try again. If it still fails, go to the [Nebula Graph community](#) for help.

8.2.5 View snapshots

To view all existing snapshots, run `SHOW SNAPSHOT`.

```
nebula> SHOW SNAPSHOT;
+-----+-----+-----+
| Name | Status | Hosts |
+-----+-----+-----+
```

```
+-----+-----+-----+
| "SNAPSHOT_2021_03_09_08_43_12" | "VALID" | "127.0.0.1:9779" |
+-----+-----+-----+
| "SNAPSHOT_2021_03_09_09_10_52" | "VALID" | "127.0.0.1:9779" |
+-----+-----+-----+
```

The parameters in the return information are described as follows.

| Parameter | Description |
|-----------|---|
| Name | Name of the snapshot directory. |
| Status | Status of the snapshot. <code>VALID</code> indicates that the creation succeeded and <code>INVALID</code> indicates that it failed. |
| Hosts | IP addresses and ports of all Storage servers at the time the snapshot was created. |

8.2.6 Delete a snapshot

To delete a snapshot, use the following syntax:

```
DROP SNAPSHOT <snapshot_name>;
```

Example:

```
nebula> DROP SNAPSHOT SNAPSHOT_2021_03_09_08_43_12;
nebula> SHOW SNAPSHTOS;
+-----+-----+-----+
| Name      | status | Hosts      |
+-----+-----+-----+
| "SNAPSHOT_2021_03_09_09_10_52" | "VALID" | "127.0.0.1:9779" |
+-----+-----+-----+
```

8.2.7 Restore data with a snapshot

1. [Find the snapshot directories](#) you want to use for data restoration.

2. Choose an approach to restore the data files:

- Change the `data_path` in the [Meta configuration](#) and [Storage configuration](#) to the snapshot path.
- Copy the snapshot directories to other locations, and change the `data_path` to these locations.
- Copy all the content in the snapshot directories into the directories where the `checkpoints` directories are located, and cover the existing files that have duplicate names with them. For example, cover `/usr/local/nebula/data/meta/nebula/0/data` with `/usr/local/nebula/data/meta/nebula/0/checkpoints/SNAPSHOT_2021_03_09_09_10_52/data`.

3. [Restart Nebula Graph](#).

8.2.8 Another way to backup and restore data

You can also use [Backup&Restore](#) to backup and restore Nebula Graph data. (TODO: coding)

Last update: April 22, 2021

9. Service Tuning

9.1 Compaction

This document gives some information about compaction.

9.1.1 Introduction to compaction

In Nebula Graph, compaction is the most important background process. Compaction has an important effect on performance.

Compaction reads the data that is written on the hard disk, then re-organizes the data structure and the indexes to make the data easier to read. The read performance can increase by times after compaction. Thus, to get high read performance, trigger compaction manually when writing a large amount of data into Nebula Graph. Note that compaction leads to long time hard disk IO, we suggest that you do compaction during off-peak hours (for example, early morning).

Nebula Graph has two types of compaction: automatic compaction and full compaction.

9.1.2 Automatic compaction

Automatic compaction is done when the system reads data, writes data, or the system restarts. The automatic compaction is enabled by default. But once triggered during peak hours, it can cause unexpected IO occupancy that has an unwanted effect on the performance. To disable automatic compaction, use this statement:

```
nebula> UPDATE CONFIGS storage:rocksdb_column_family_options = {disable_auto_compactions = true};
```

⚠ Caution

The command overwrites all `rocksdb_column_family_options` items. Other items besides `disable_auto_compactions` is overwritten to the default value. You may have to read all the items before the updates.

9.1.3 Full compaction

Full compaction enables large scale background operations for a graph space such as merging files, deleting the data expired by TTL. Use these statements to enable full compaction:

```
nebula> USE <your_graph_space>;
nebula> SUBMIT JOB COMPACT;
```

The preceding statement returns a job_id. To show the compaction progress, use this statement:

```
nebula> SHOW JOB <job_id>;
```

>Note

Do the full compaction during off-peak hours because full compaction has a lot of IO operations.

9.1.4 Operation suggestions

These are some operation suggestions to keep Nebula Graph performing well.

- To avoid unwanted IO waste during data writing, set `disable_auto_compactions` to `true` before large amounts of data writing.
- After data import is done, run `SUBMIT JOB COMPACT`.
- Run `SUBMIT JOB COMPACT` periodically during off-peak hours, for example, early morning.
- Set `disable_auto_compactions` to `false` during day time.
- To control the read and write traffic limitation for compactions, set these two parameters in the `nebula-storaged.conf` configuration file.

```
# read from the local configuration file and start
--local-config=true
--rate_limit=20 (in MB/s)
```

9.1.5 FAQ

Q: Can I do full compactions for multiple graph spaces at the same time? A: Yes, you can. But the IO is much larger at this time.

Q: How much time does it take for full compactions? A: When `rate_limit` is set to `20`, you can estimate the full compaction time by dividing the hard disk usage by the `rate_limit`. If you do not set the `rate_limit` value, the empirical value is around 50 MB/s.

Q: Can I modify `--rate_limit` dynamically? A: No, you cannot.

Q: Can I stop a full compaction after it starts? A: No you cannot. When you start a full compaction, you have to wait till it is done. This is the limitation of RocksDB.

Last update: April 22, 2021

9.2 Storage load balance

You can use the `BALANCE` statements to balance the distribution of partitions and Raft leaders, or remove redundant Storage servers.

9.2.1 Prerequisites

The graph spaces stored in Nebula Graph must have more than one replicas for the system to balance the distribution of partitions and Raft leaders.

9.2.2 Balance partition distribution

`BALANCE DATA` starts a task to equally distribute the storage partitions in a Nebula Graph cluster. A group of subtasks will be created and implemented to migrate data and balance the partition distribution.

Danger

DON'T stop any machine in the cluster or change its IP address until all the subtasks finish. Otherwise, the follow-up subtasks fail.

Take scaling out Nebula Graph for an example.

After you add new storage hosts into the cluster, no partition is deployed on the new hosts. You can run `SHOW HOSTS` to check the partition distribution.

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 4 | "basketballplayer:4" | "basketballplayer:15" |
+-----+-----+-----+-----+-----+
| "storaged1" | 9779 | "ONLINE" | 8 | "basketballplayer:8" | "basketballplayer:15" |
+-----+-----+-----+-----+-----+
| "storaged2" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:15" |
+-----+-----+-----+-----+-----+
| "storaged3" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" |
+-----+-----+-----+-----+-----+
| "storaged4" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" |
+-----+-----+-----+-----+-----+
| "Total" | | 15 | "basketballplayer:15" | "basketballplayer:45" |
+-----+-----+-----+-----+-----+
Got 6 rows (time spent 1002/1780 us)
```

Run `BALANCE DATA` to start balancing the storage partitions. If the partitions are already balanced, `BALANCE DATA` fails.

```
nebula> BALANCE DATA;
+-----+
| ID |
+-----+
| 1614237867 |
+-----+
Got 1 rows (time spent 3783/4533 us)
```

A `BALANCE` task ID is returned after running `BALANCE DATA`. Run `BALANCE DATA <balance_id>` to check the status of the `BALANCE` task.

```
nebula> BALANCE DATA 1614237867;
+-----+-----+
| balanceId, spaceId:partId, src->dst | status |
+-----+-----+
| "[1614237867, 11:1, storaged1:9779->storaged3:9779]" | "SUCCEEDED" |
+-----+-----+
| "[1614237867, 11:1, storaged2:9779->storaged4:9779]" | "SUCCEEDED" |
+-----+-----+
| "[1614237867, 11:2, storaged1:9779->storaged3:9779]" | "SUCCEEDED" |
+-----+-----+
...
+-----+-----+
| "Total:22, Succeeded:22, Failed:0, In Progress:0, Invalid:0" | 100 |
+-----+-----+
Got 23 rows (time spent 916/1528 us)
```

When all the subtasks succeed, the load balancing process finishes. Run `SHOW HOSTS` again to make sure the partition distribution is balanced.

Note

`BALANCE DATA` does not balance the leader distribution.

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 4 | "basketballplayer:4" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged1" | 9779 | "ONLINE" | 8 | "basketballplayer:8" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged2" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged3" | 9779 | "ONLINE" | 0 | "No valid partition" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged4" | 9779 | "ONLINE" | 0 | "No valid partition" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "Total" | | 15 | "basketballplayer:15" | "basketballplayer:45" |
+-----+-----+-----+-----+-----+
Got 6 rows (time spent 849/1420 us)
```

If any subtask fails, run `BALANCE DATA` again to restart the balancing. If redoing load balancing does not solve the problem, ask for help in the [Nebula Graph community](#).

9.2.3 Stop data balancing

To stop a balance task, run `BALANCE DATA STOP`.

- If no balance task is running, an error is returned.
- If a balance task is running, the task ID is returned.

`BALANCE DATA STOP` does not stop the running subtasks but cancels all follow-up subtasks. The running subtasks continue.

To check the status of the stopped balance task, run `BALANCE DATA <balance_id>`.

Once all the subtasks are finished or stopped, you can run `BALANCE DATA` again to balance the partitions again.

- If any subtask of the preceding balance task failed, Nebula Graph restarts the preceding balance task.
- If no subtask of the preceding balance task failed, Nebula Graph starts a new balance task.

9.2.4 Remove storage servers

To remove specific storage servers and scale in the Storage Service, use the `BALANCE DATA REMOVE <host_list>` syntax.

For example, to remove the following storage servers:

| Server name | IP | Port |
|-------------|-------------|-------|
| storage3 | 192.168.0.8 | 19779 |
| storage4 | 192.168.0.9 | 19779 |

Run the following statement:

```
BALANCE DATA REMOVE 192.168.0.8:19779,192.168.0.9:19779;
```

Nebula Graph will start a balance task, migrate the storage partitions in storage3 and storage4, and then remove them from the cluster.

Note

The removed server's state will change to `OFFLINE`.

9.2.5 Balance leader distribution

`BALANCE DATA` only balances the partition distribution. If the raft leader distribution is not balanced, some of the leaders may overload. To load balance the raft leaders, run `BALANCE LEADER`.

```
nebula> BALANCE LEADER;
Execution succeeded (time spent 7576/8657 us)
```

Run `SHOW HOSTS` to check the balance result.

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged1" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged2" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged3" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "storaged4" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:9" |
+-----+-----+-----+-----+-----+
| "Total" | | | 15 | "basketballplayer:15" | "basketballplayer:45" |
+-----+-----+-----+-----+-----+
```

Last update: May 10, 2021

10. Nebula Graph Dashboard

10.1 What is Nebula Graph Dashboard

Nebula Graph Dashboard (Dashboard for short) is a visualization tool that monitors the status of machines and services in Nebula Graph clusters.

10.1.1 Features

Dashboard monitors:

- The status of all the machines in clusters, including CPU, memory, load, disk, and network.
- The information of all the services in clusters, including the IP addresses, versions, and monitoring metrics (such as the number of queries, the latency of queries, the latency of heartbeats, and so on).
- The information of clusters, including the information of services, partitions, configurations, and long-term tasks.

Features of the enterprise package (TODO: planning)

Enterpriseonly

Dashboard has two editions. One is open source, the other is for enterprises. Most of the functions are available in both. The functions only supported by the enterprise package will be marked and explained.

10.1.2 Scenarios

You can use Dashboard in one of the following scenarios:

- You want to monitor key metrics conveniently and quickly, and present multiple key information of the business to ensure the business operates normally.
- You want to monitor clusters from multiple dimensions (such as the time, aggregate rules, and metrics).
- After a failure occurs, you need to review it and confirm its occurrence time and unexpected phenomena.

10.1.3 Precautions

- The monitoring data will be updated per 7 seconds by default.
- The monitoring data will be retained for 14 days by default, that is, only the monitoring data within the last 14 days can be queried.

Note

The monitoring service is supported by Prometheus. The update frequency and retention intervals can be modified. For details, see [Prometheus](#).

Last update: July 27, 2021

10.2 Deploy Dashboard

The deployment of Dashboard involve five services. This topic will describe how to deploy Dashboard in detail.

10.2.1 Nebula Graph releases

The correspondence between the Dashboard release and the Nebula Graph release is as follows.

| Dashboard | Nebula Graph |
|-----------------------------|--------------|
| nebula-graph-dashboard-beta | 2.5.0, 2.0.1 |

10.2.2 Port

The deployment of Dashboard occupies the following ports:

- 9200
- 9100
- 9090
- 8090
- 7003

10.2.3 Download Dashboard

Download the configuration files for the deployment.

```
wget https://oss-cdn.nebula-graph.com.cn/nebula-graph-dashboard/nebula-graph-dashboard-beta.tar.gz
```

10.2.4 Service

Run `tar -xvf nebula-graph-dashboard-beta.tar.gz` to decompress the installation package. There are 5 services in the `nebula-graph-dashboard`. The descriptions are as follows.

| Name | Description |
|------------------------|--|
| node-exporter | Collects the source information of machines in the cluster, including the CPU, memory, load, disk, and network. |
| nebula-stats-exporter | Collects the performance metrics in the cluster, including the IP addresses, versions, and monitoring metrics (such as the number of queries, the latency of queries, the latency of heartbeats, and so on). |
| prometheus | The time series database that stores monitoring data. |
| nebula-http-gateway | Provides HTTP ports for cluster services to access the prometheus service or execute nGQL statements to interact with the Nebula Graph database. |
| nebula-graph-dashboard | Provides the Dashboard service. Note that its name is the same as its superordinate. The following <code>nebula-graph-dashboard</code> refers to this service. |

The above five services should be deployed as follows.

10.2.5 Procedure

Deploy node-exporter

Note

You need to deploy the `node-exporter` service on each machine in the cluster.

To start the service, run the following statement in `node-exporter`:

```
$ nohup ./node-exporter --web.listen-address=:9100 &
```

After the service is started, you can enter `<IP>:9100` in the browser to check whether the service is started normally.

Deploy nebula-stats-exporter

Note

You only need to deploy the `nebula-stats-exporter` service on the machine where the `nebula-graph-dashboard` service is installed.

1. Modify the `config.yaml` file in `nebula-stats-exporter` to deploy the HTTP ports of all the services. The example is as follows:

```
version: v0.0.2
nebulaItems:
  - instanceName: metad0
    endpointIP: 192.168.xx.1
    endpointPort: 19559
    componentType: metad
  - instanceName: metad1
    endpointIP: 192.168.xx.2
    endpointPort: 19559
    componentType: metad
  - instanceName: metad2
    endpointIP: 192.168.xx.3
    endpointPort: 19559
    componentType: metad
  - instanceName: graphd0
    endpointIP: 192.168.xx.4
    endpointPort: 19669
    componentType: graphd
  - instanceName: storaged0
    endpointIP: 192.168.xx.5
    endpointPort: 19779
    componentType: storaged
  - instanceName: storaged1
    endpointIP: 192.168.xx.6
    endpointPort: 19779
    componentType: storaged
  - instanceName: storaged2
    endpointIP: 192.168.xx.7
    endpointPort: 19779
```

2. Run the following statement to start the service:

```
$ nohup ./nebula-stats-exporter --bare-metal --bare-metal-config=../config.yaml &
```

After the service is started, you can enter `<IP>:9200` in the browser to check whether the service is started normally.

Deploy prometheus

Note

You only need to deploy the `prometheus` service on the machine where the `nebula-graph-dashboard` service is installed.

1. Modify the `prometheus.yaml` file in `prometheus` to deploy the IP addresses and ports of the `node-exporter` service and the `nebula-stats-exporter`. The example is as follows:

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s
scrape_configs:
  - job_name: 'nebula-exporter'
    static_configs:
      - targets: [
        '192.168.xx.100:9200', # The IP address and port of the nebula-stats-exporter service.
      ]
  - job_name: 'node-exporter'
    static_configs:
      - targets: [
        '192.168.xx.101:9100' # The IP address and port of the node-exporter service.
      ]
```

- `scrape_interval`: The interval for collecting the monitoring data, which is 1 minute by default.
- `evaluation_interval`: The interval for running alarm rules, which is 1 minute by default.

2. Run the following statement to start the service.

```
$ nohup ./prometheus --config.file= ./prometheus.yaml &
```

After the service is started, you can enter `<IP>:9090` in the browser to check whether the service is started normally.

Deploy `nebula-http-gateway`

Note

You only need to deploy the `nebula-http-gateway` service on the machine where the `nebula-graph-dashboard` service is installed.

To start the service, run the following statement in `nebula-http-gateway`:

```
$ nohup ./nebula-httdp &
```

After the service is started, you can enter `<IP>:8090` in the browser to check whether the service is started normally.

How to deploy the `nebula-graph-dashboard` service

1. Modify the `custom.json` file in `nebula-graph-dashboard/static/` to deploy the IP address and port of the Graph Service. The example is as follows:

```
{
  "connection": {
    "ip": "192.168.xx.4",
    "port": 9669
  },
  "alias": {
    "ip:port": "instance1"
  },
  "chartBaseLine": {
  }
}
```

2. To start the service, run the following statement in `nebula-graph-dashboard`:

```
$ npm run start
```

After the service is started, you can enter `<IP>:7003` in the browser to check whether the service is started normally.

10.2.6 Stop Dashboard

You can enter `kill <pid>` to stop Dashboard. The examples are as follows:

```
$ kill $(lsof -t -i :9100) # stop the node-exporter service
$ kill $(lsof -t -i :9200) # stop the nebula-stats-exporter service
$ kill $(lsof -t -i :9090) # stop the prometheus service
$ kill $(lsof -t -i :8090) # stop the nebula-http-gateway service
$ cd nebula-graph-dashboard
$ npm run stop # stop the nebula-graph-dashboard service
```

.....

Last update: August 19, 2021

10.3 Connect Dashboard

After Dashboard is deployed, you can log in and use Dashboard on the browser.

10.3.1 Prerequisites

- The Dashboard services are started. For more information, see [Deploy Dashboard](#).
- We recommend you to use the Chrome browser of the version above 58. Otherwise, there may be compatibility issues.

10.3.2 Procedure

1. Confirm the IP address of the machine where the `nebula-graph-dashboard` service is installed. Enter `<IP>:7003` in the browser to open the login page.
 2. Enter the username and the passwords of the Nebula Graph database and click the login button.
 - If authentication is enabled, you can log in with the created accounts.
 - If authentication is not enabled, you can only log in using `root` as the username and random characters as the password.
- To enable authentication, see [Authentication](#)



Nebula Dashboard

Account Login

root

.....

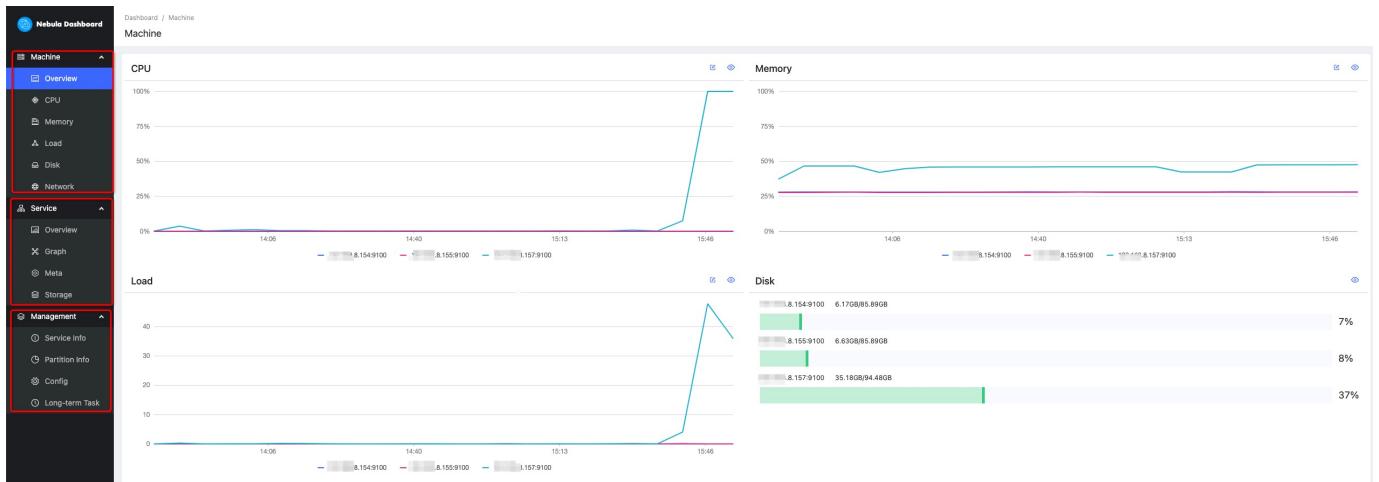
Login

Last update: July 27, 2021

10.4 Dashboard

Nebula Dashboard consists of three parts: Machine, Service, and Management. This topic will describe them in detail.

10.4.1 Overview



10.4.2 Machine

Machine consists of the following parts:

- Overview

You can check the fluctuations of CPU, Memory, Load, Disk, Network In, and Network Out in the past 24 hours.

For details of certain monitoring metrics, you can click the  symbol in the upper right corner, or click the monitoring metrics on the left.

- CPU, Memory, Load, Disk, Network

It shows the detailed monitoring data of the machine from the above dimensions.

- By default, you can check the monitoring data up to 14 days before. The alternative can be 1 hour, 6 hours, 12 hours, 1 day, 3 days, 7 days, or 14 days in the past.
- You can choose the machine and monitoring metrics that you want to check. For more information, see [monitor parameter](#).
- You can set a base line as a reference.



10.4.3 Service

Service consists of the following parts:

- Overview

You can check the fluctuations of monitoring metrics of various services in the past 24 hours. You can also switch to the **Version** page to view the IP addresses and versions of all services.



For details of certain monitoring metrics, you can click the  symbol in the upper right corner, or click the services on the left.

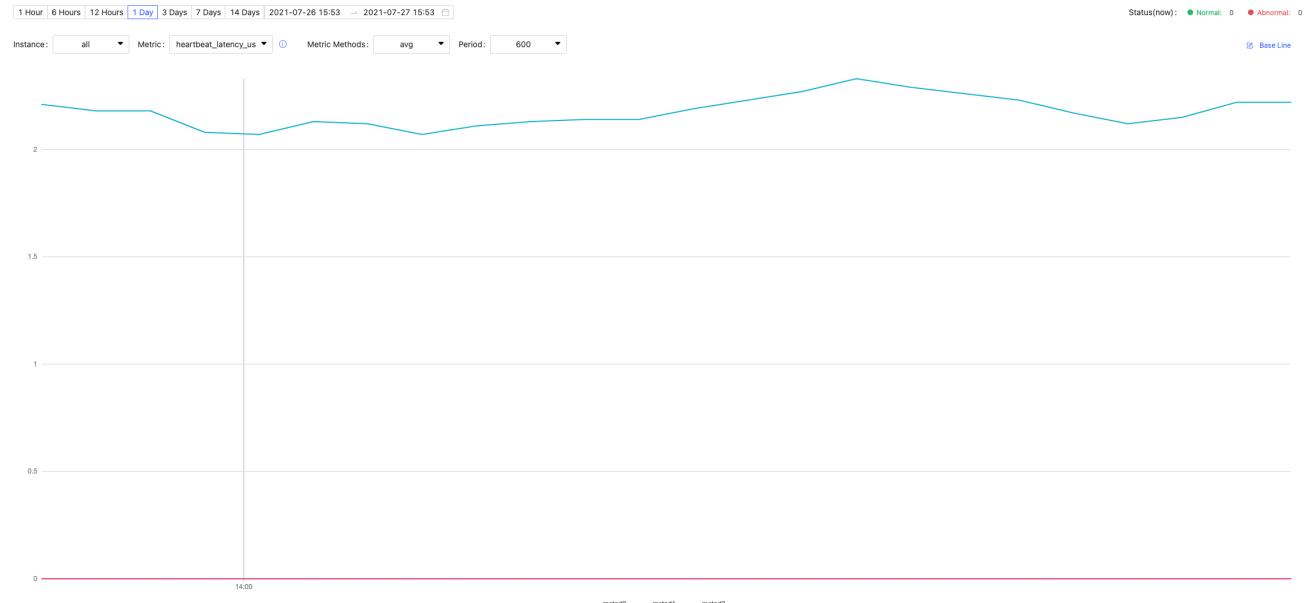
Note

The overview page of the current open source edition only supports setting two monitoring metrics for each service. You can adjust it by clicking the **Set up** button.

- Graph, Meta, Storage

It shows the detailed monitoring data of the above services.

- By default, you can check the monitoring data up to 14 days before. The alternative can be 1 hour, 6 hours, 12 hours, 1 day, 3 days, 7 days, or 14 days in the past.
- You can choose the machine that you want to check the monitoring data, monitoring metrics, metric methods, and period. For more information, see [monitor parameter](#)
- You can set a base line as a reference.
- You can check the status of the current service.



10.4.4 Management

Note

Non-root users can view the service information and the partition information with spatial permissions, but cannot view the configuration and long-term tasks.

Management consists of the following parts:

- **Service Info**

It shows the basic information of the Storage Service, including the information of the host, the number of leaders, the distribution of partitions, and the distribution of leaders.

- **Partition Info**

You can check the information of partitions in different graph spaces. The descriptions are as follows.

| Parameter | Description |
|--------------|--|
| Partition ID | The ID of the partition. |
| Leader | The IP address and the port of the leader. |
| Peers | The IP addresses and the ports of all the replicas. |
| Losts | The IP addresses and the ports of replicas at fault. |

- **Config**

It shows the configuration of each service. Dashboard does not support online modification of configurations for now. For details, see [configurations](#).

- **Long-term Task**

It shows the information of all jobs. Dashboard does not support online management of jobs for now. For details, see [job statements](#).

10.4.5 Others

In the lower left corner of the page, you can:

- Sign out
- Switch between Chinese and English
- View the current Dashboard release
- View the user manual and forum
- Fold the sidebar

Last update: August 19, 2021

10.5 Metrics

This topic will describe the monitoring metrics in Nebula Graph Dashboard.

Note

The default unit in **Disk** and **Network** is byte. The unit will change with the data magnitude as the page displays. For example, when the flow is less than 1 KB/s, the unit will be Bytes/s.

10.5.1 Machine

CPU

| Parameter | Description |
|-----------------|--|
| cpu_utilization | The percentage of used CPU. |
| cpu_idle | The percentage of idled CPU. |
| cpu_wait | The percentage of CPU waiting for IO operations. |
| cpu_user | The percentage of CPU used by users. |
| cpu_system | The percentage of CPU used by the system. |

Memory

| Parameter | Description |
|--------------------|---|
| memory_utilization | The percentage of used memory. |
| memory_used | The memory space used (including caches). |
| memory_actual_used | The memory space used (not including caches). |
| memory_free | The memory space available. |

Load

| Parameter | Description |
|-----------|--|
| load_1m | The average load of the system in the last 1 minute. |
| load_5m | The average load of the system in the last 5 minutes. |
| load_15m | The average load of the system in the last 15 minutes. |

Disk

| Parameter | Description |
|-------------------|--|
| disk_used | The disk space used. |
| disk_free | The disk space available. |
| disk_readbytes | The number of bytes that the system reads in the disk per second. |
| disk_writebytes | The number of bytes that the system writes in the disk per second. |
| disk_readiops | The number of read queries that the disk receives per second. |
| disk_writeiops | The number of write queries that the disk receives per second. |
| inode_utilization | The percentage of used inode. |

Network

| Parameter | Description |
|---------------------|---|
| network_in_rate | The number of bytes that the network card receives per second. |
| network_out_rate | The number of bytes that the network card sends out per second. |
| network_in_errs | The number of wrong bytes that the network card receives per second. |
| network_out_errs | The number of wrong bytes that the network card sends out per second. |
| network_in_packets | The number of data packages that the network card receives per second. |
| network_out_packets | The number of data packages that the network card sends out per second. |

10.5.2 Service**Period**

The period is the time range of counting metrics. It currently supports 5 seconds, 60 seconds, 600 seconds, and 3600 seconds, which respectively represent the last 5 seconds, the last 1 minute, the last 10 minutes, and the last 1 hour.

Metric methods

| Parameter | Description |
|-----------|--|
| rate | The average rate of operations per second in a period. |
| sum | The sum of operations in the period. |
| avg | The average latency in the cycle. |
| P75 | The 75th percentile latency. |
| P95 | The 95th percentile latency. |
| P99 | The 99th percentile latency. |
| P999 | The 99.9th percentile latency. |

Graph

| Parameter | Description |
|-----------------------|--------------------------------------|
| num_queries | The number of queries. |
| num_slow_queries | The number of slow queries. |
| query_latency_us | The average latency of queries. |
| slow_query_latency_us | The average latency of slow queries. |
| num_query_errors | The number of queries in error. |

Meta

| Parameter | Description |
|----------------------|----------------------------|
| heartbeat_latency_us | The latency of heartbeats. |
| num_heartbeats | The number of heartbeats. |

Storage

| Parameter | Description |
|----------------------------|--|
| add_edges_latency_us | The average latency of adding edges. |
| add_vertices_latency_us | The average latency of adding vertices. |
| delete_edges_latency_us | The average latency of deleting edges. |
| delete_vertices_latency_us | The average latency of deleting vertices. |
| forward_tranx_latency_us | The average latency of transmitting. |
| get_neighbors_latency_us | The average latency of querying neighbors. |

Last update: August 19, 2021

11. Nebula Spark Connector

Nebula Spark Connector is a Spark connector application for reading and writing Nebula Graph data in Spark standard format. Nebula Spark Connector consists of two parts: Reader and Writer.

- Reader

Provides a Spark SQL interface. This interface can be used to read Nebula Graph data. It reads one vertex or edge type data at a time and assemble the result into a Spark DataFrame.

- Writer

Provides a Spark SQL interface. This interface can be used to write DataFrames into Nebula Graph in a row-by-row or batch-import way.

For more information, see [Nebula Spark Connector](#).

11.1 Use cases

Nebula Spark Connector applies to the following scenarios:

- Migrate data between different Nebula Graph clusters.
- Migrate data between different graph spaces in the same Nebula Graph cluster.
- Migrate data between Nebula Graph and other data sources.

11.2 Benefits

- Supports multiple connection settings, such as timeout period, number of connection retries, number of execution retries, etc.
- Supports multiple settings for data writing, such as setting the corresponding column as vertex ID, starting vertex ID, destination vertex ID or attributes.
- Supports non-attribute reading and full attribute reading.
- Supports reading Nebula Graph data into VertexRDD and EdgeRDD, and supports non-Long vertex IDs.
- Nebula Spark Connector 2.0 unifies the extended data source of SparkSQL, and uses DataSourceV2 to extend Nebula Graph data.

Last update: May 20, 2021

12. Nebula Flink Connector

Nebula Flink Connector is a connector that helps Flink users quickly access Nebula Graph. Nebula Flink Connector supports reading data from the Nebula Graph database or writing other external data to the Nebula Graph database.

For more information, see [Nebula Flink Connector](#).

12.1 Use cases

Nebula Flink Connector applies to the following scenarios:

- Migrate data between different Nebula Graph clusters.
 - Migrate data between different graph spaces in the same Nebula Graph cluster.
 - Migrate data between Nebula Graph and other data sources.
-

Last update: May 20, 2021

13. Contribution

13.1 How to Contribute

13.1.1 Before you get started

File an issue on the github or forum

You are welcome to contribute any code or files to the project. But first we suggest you raise an issue on the [github](#) or on the [forum](#) to start a discussion with the community. Check through the topic for Github.

Sign the Contributor License Agreement (CLA)

What is [CLA](#)?

Here is the [vesoft inc. Contributor License Agreement](#).

Click the **Sign in with GitHub to agree** button to sign the CLA.

If you have any question, send an email to info@vesoft.com.

13.1.2 Step 1: Fork in the github.com

The Nebula Graph project has many [repositories](#). Take [the graph engine repository](#) for example:

1. Visit <https://github.com/vesoft-inc/nebula-graph>
2. Click the `Fork` button (top right) to establish an online fork.

13.1.3 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-graph.git
# the following is recommended
# or: git clone git@github.com:$user/nebula-graph.git

cd $working_dir/nebula
git remote add upstream https://github.com/vesoft-inc/nebula-graph.git
# or: git remote add upstream git@github.com:vesoft-inc/nebula-graph.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:$(user)/nebula-graph.git (fetch)
# origin  git@github.com:$(user)/nebula-graph.git (push)
# upstream https://github.com/vesoft-inc/nebula-graph (fetch)
# upstream no_push (push)
git remote -v
```

Define a Pre-Commit Hook

Please link the **Nebula Graph** pre-commit hook into your `.git` directory.

This hook checks your commits for formatting, building, doc generation, etc.

```
cd $working_dir/nebula-graph/.git/hooks
ln -s $working_dir/nebula-graph/.linters/cpp/hooks/pre-commit.sh .
```

Sometimes, pre-commit hook can not be executable. You have to make it executable manually.

```
cd $working_dir/nebula-graph/.git/hooks
chmod +x pre-commit
```

13.1.4 Step 3: Branch

Get your local master up to date:

```
cd $working_dir/nebula-graph
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
```

13.1.5 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 bug fixes.

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-graph`, run the following command:

```
ctest -j$(nproc)
```

13.1.6 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.

13.1.7 Step 6: Commit

Commit your changes.

```
git commit -a
```

Likely you'll go back and edit/build/test some more than `--amend` in a few cycles.

13.1.8 Step 7: Push

When ready to review (or just to establish an offsite backup of your work), push your branch to your fork on `github.com`:

```
git push origin myfeature
```

13.1.9 Step 8: Create a Pull Request

1. Visit your fork at [https://github.com/\\$user/nebula-graph](https://github.com/$user/nebula-graph) (replace `$user` obviously).
2. Click the `Compare & pull request` button next to your `myfeature` branch.

13.1.10 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 22, 2021

14. FAQ

14.1 FAQ

This topic lists the frequently asked questions for using Nebula Graph. You can use the search box in the help center or the search function of the browser to match the questions you are looking for.

If the solutions described in this topic cannot solve the problem, ask for help on the [Nebula Graph forum](#) or submit an issue on [GitHub](#).

14.1.1 About manual updates

?

Why is the behavior of manual not consistent with the system?

Nebula Graph 2.0 is still under development. Its behavior changes from time to time. Please tell us if the manual and the system are not consistent.

?

Some errors in this manula

1. Click the `pencil` button at the top right side of this page.
2. Use markdown to fix this error. Then "Commit changes" at the bottom, which will start a Github pull request.
3. Sign the [CLA](#). This pull request (and the fix) will be merged after to reviewer's accept.

14.1.2 About forward and backward compatibility

⚠ Major version compatibility

Neubla Graph 2.5.0 is not compatible with Nebula Graph 1.x nor 2.0-RC in both data formats and RPC-protocols, and vice versa. Check [how to upgrade to Neubla Graph 2.5.0](#). You must upgrade [all clients](#).

?

Micro version compatibility

Neubla Graph 2.5.0 is compatible with Nebula Graph 2.0 in both data formats and RPC-protocols.

14.1.3 About executions

?

How is the time spent value at the end of each return message calculated?

Take the return message of `SHOW SPACES` as an example:

```
nebula> SHOW SPACES;
+-----+
| Name      |
+-----+
| basketballplayer |
+-----+
Got 1 rows (time spent 1235/1934 us)
```

- The first number 1235 shows the time spent by the database itself, that is, the time it takes for the query engine to receive a query from the client, fetch the data from the storage server and perform a series of calculations.
- The second number 1934 shows the time spent from the client's perspective, that is, the time it takes for the client from sending a request, receiving a response, and displaying the result on the screen.

?

Can I set `replica_factor` as an even number in `CREATE SPACE` statements, e.g., `replica_factor = 2`?

NO.

The Storage Service guarantees its availability based on the Raft consensus protocol. The number of failed replicas must not exceed half of the total replica number.

When `replica_factor=2`, if one replica fails, the Storage Service fails. No matter `replica_factor=3` or `replica_factor=4`, if more than one replica fails, the Storage Service fails, so `replica_factor=3` is recommended.

To prevent unnecessary waste of resources, we recommend that you set an odd replica number.

We suggest that you set `replica_factor` to 3 for the production environment and 1 for the test environment. Do not use an even number.

?

How to resolve [ERROR (-7)]: SyntaxError: syntax error near ?

In most cases, a query statement requires a `YIELD` or a `RETURN`. Check your query statement to see if `YIELD` or `RETURN` is provided.

?

How to count the vertices/edges number of each tag/edge type?

See [show-stats](#).

?

How to get all the vertices/edge of each tag/edge type?

1. Create and rebuild the index.

```
> CREATE TAG INDEX i_player ON player();
> REBUILD TAG INDEX i_player;
```

2. Use `LOOKUP` or `MATCH`.

```
> LOOKUP ON player;
> MATCH (n:player) RETURN n;
```

See [INDEX](#), [LOOKUP](#) and [MATCH](#).

?

How to resolve the can't solve the start vids from the sentence error?

The graphd requires `start vids` to begin a graph traversal. The `start vids` can either be specified by the user, for example,

```
> GO FROM ${vids} ...
> MATCH (src) WHERE id(src) == ${vids}
# The start vids are explicitly given by ${vids}.
```

or be found from a (property) index, for example,

```
# CREATE TAG INDEX i_player ON player(name(20));
# REBUILD TAG INDEX i_player;

> LOOKUP ON player WHERE player.name == "abc" | ... YIELD ...
> MATCH (src) WHERE src.name == "abc" ...
# The start vids are found from the property index on name.
```

Otherwise, an error like `can't solve the start vids from the sentence` will be raised.

?

How to resolve error Storage Error: The VID must be a 64-bit integer or a string. ?

Check your vid is an integer or a `fix_string(N)`. If it is a string type, make sure your input is not longer than `N` (default value is `8`). See [create space](#).

?

How to resolve error edge conflict or vertex conflict ?

Nebula Graph returns such errors when the Storage Service receives multiple requests to insert or update the same vertex or edge within milliseconds. Try the failed requests again later.

?

Storage Error E_RPC_FAILURE

Storage returns too many data back to graphd. Possible solutions:

1. Check whether storage is Out-of-memory. (`dmesg | grep nebula`).
2. In `nebula-graphd.conf`, modify or add the item `--storage_client_timeout_ms=60000` to change the timeout(ms).
3. Modify your nGQL to reduce full scans (including limit sentence).
4. Use better hardware (NVMe, more memory).
5. retry

?

The leader has changed. Try again later

Known Issue. Just retry 1 to N times, where N is the partition number. The reason is that meta client needs some heartbeats to update or errors to trigger the new leader information.

?

How to stop a slow query

You can't. Even killing a client will not stop the running slow query. You have to wait the query to complete.

14.1.4 About operation and maintenance

?

The log files are too large. How to recycle the logs?

Nebula Graph uses `glog` to print logs. `glog` can't recycle the outdated files. You can use crontab to delete them by yourself. Refer to the discussions of [Glog should delete old log files automatically](#).

?

How to check the Nebula Graph version?

1. Use the `<binary_path> --version` command to get the Git commit IDs of the Nebula Graph binary files.

For example, to check the version of the Graph Service, go to the directories where the `nebula-graphd` binary files are stored, and run `./nebula-graphd --version` as follows to get the commit IDs.

```
$ ./nebula-graphd --version
nebula-graphd version Git: ab4f683, Build Time: Mar 24 2021 02:17:30
This source code is licensed under Apache 2.0 License, attached with Common Clause Condition 1.0.
```

2. Search for the commit ID obtained in the preceding step on the [GitHub commits](#) page.

3. Compare the commit time of the binary files with the [release time](#) of Nebula Graph versions to find out the version of the Nebula Graph services.

?

How to scale out or scale in?

Nebula Graph 2.5.0 doesn't provide any commands or tools to support automatic scale out/in. You can do by the following steps

1. `metad` `metad` can not be scaled out or scale in. The process can't be moved to a new machine. You can not add a new `metad` process to the service.
2. Scale in `graphd` `remove graphd` from client's code. Close this `graphd` process.
3. Scale out `graphd` `prepare graphd's binary and config files in the new host. Modify the config files and add all existing metad's addresses. Then start the new graphd process.`
4. Scale in `storaged` `All spaces' replace number must be greater than 1` [ref to Balance remove command](#). After the command finish, stop this `storaged` process.
5. Scale out `storaged` `All spaces' replace number must be greater than 1` `prepare storaged's binary and config files in the new host. Modify the config files and add all existing metad's adDresses. Then start the new storaged process.`

You may also need to run [Balance Data](#) and [Balance leader](#) after scaling in/out storaged.

14.1.5 About connections

?

Which ports should be opened on the firewalls?

If you have not changed the predefined ports in the [configurations](#), open the following ports for the Nebula Graph services:

| Service | Ports |
|---------|---------------------------|
| Meta | 9559, 9560, 19559, 19560 |
| Graph | 9669, 19669, 19670 |
| Storage | 9777 ~ 9780, 19779, 19780 |

If you have customized the configuration files and changed the predefined ports, find the port numbers in your configuration files and open them on the firewalls.

❓ How to test whether a port is open or closed?

You can use telnet as follows to check for port status.

```
telnet <ip> <port>
```

For example:

```
// If the port is open:  
$ telnet 192.168.1.10 9669  
Trying 192.168.1.10...  
Connected to 192.168.1.10.  
Escape character is '^]'.  
  
// If the port is closed or blocked:  
$ telnet 192.168.1.10 9777  
Trying 192.168.1.10...  
telnet: connect to address 192.168.1.10: Connection refused
```

If you cannot use the telnet command, check if telnet is installed or enabled on your host.

Last update: July 7, 2021

15. Appendix

15.1 VID

TODO(doc)

15.1.1 VID

`VID` is short for vertex identifier.

In Nebula Graph, vertices are identified with vertex identifiers (i.e. `VID`s). The VID can be an int64 or a fixed length string. When inserting a vertex, you must specify a `VID` for it.

You can also call `hash()` to generate an int64 VID if the graph has less than one billion vertices.

`VID` must be unique in a graph space.

That is, in the same graph space, two vertices that have the same `VID` are considered as the same vertex.

In addition, one `VID` can have multiple `TAG`s. E.g., One person (`VID`) can have two roles (`tags`).

Two `VID`s in two different graph spaces are totally independent of each other.

Last update: May 20, 2021

15.2 Graph modeling

15.2.1 Graph data modeling suggestions

This section provides general suggestions for modeling data in Nebula Graph.

Note

The following suggestions may not apply to some special scenarios. In these cases, find help in the [Nebula Graph community](#).

Model for performance

There is no perfect method to model in Nebula Graph. Graph modeling depends on the questions that you want to know from the data. Your data drives your graph model. Graph data modeling is intuitive and convenient. Create your data model based on your business model. Test your model and gradually optimize it to fit your business. To get better performance, you can change or redesign your model multiple times.

Edges as properties

Traversal depth decreases the traversal performance. To decrease the traversal depth, use vertex properties instead of edges.

For example, to model a graph that have the name, age, and eye color elements, you can:

- (RECOMMENDED) Create a tag `person`, then add the name, age, and eye color as its properties.
- (WRONG WAY) Create a new tag `eye color` and a new edge type `has`, then create an edge to indicate that a person has an eye color.

The first modeling solution leads to much better performance. DO NOT use the second solution unless you have to.

Multiple properties under one tag are permitted. But make sure that tags are fine-grained. For more information, see the [Granulated vertices](#) section.

Granulated vertices

In graph modeling, use the data models with a higher level of granularity. Put a set of parallel properties into one tag, i.e., separate different concepts.

Use indexes correctly

Correct use of indexes speeds up queries, but indexes reduce the write performance by 90% or more. **ONLY** use indexes when you locate vertices or edges by their properties.

No long string properties on edges

Be careful when you create long string properties for edges. Nebula Graph supports storing such properties on edges. But note that these properties are stored both in the outgoing edges and the incoming edges. Thus be careful with the write amplification.

Last update: May 20, 2021

15.3 System modeling

TODO(doc)

Last update: May 20, 2021

15.4 About Raft

TODO(doc)

Last update: May 20, 2021

15.5 Partition ID

When inserting into Nebula Graph, vertices and edges are distributed across different partitions. And the partitions are located on different machines. If you want 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 / code](#).

```
// If the length of the id is 8, we will treat it as int64_t to be compatible
// with the version 1.0
uint64_t vid = 0;
if (id.size() == 8) {
    memcpy(static_cast<void*>(&vid), id.data(), 8);
} else {
    MurmurHash2 hash;
    vid = hash(id.data());
}
PartitionID pId = vid % numParts + 1;
```

Roughly say, after hashing a fixed string to int64, (the hashing of int64 is the number itself), do modulo and then plus one.

```
pId = vid % numParts + 1;
```

In the preceding formula,

- `%` is the modulo operation.
- `numParts` 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.
- `pId` 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.

But, the mapping between the `partition ID` and the machine address is random. Therefore, you can't assume that any two partitions are located on the same machine.

Last update: May 20, 2021

15.6 Version Description

Different versions of the ecosystem tools support different Nebula Graph kernel versions. This topic introduces the correspondence between the versions of ecosystem tools and the Nebula Graph kernel.

 **Note**

All ecosystem tools of 1.x did not support Nebula Graph 2.x kernel.

15.6.1 Nebula Studio

| Studio version | Nebula Graph version |
|----------------|----------------------|
| 2.2.0 | 2.0.0 |
| 2.2.0 | 2.0.1 |

15.6.2 Nebula Exchange

| Exchange version | Nebula Graph version |
|------------------|----------------------|
| 2.0.0 | 2.0.0 |
| 2.0.0 | 2.0.1 |

15.6.3 Nebula Importer

| Importer version | Nebula Graph version |
|------------------|----------------------|
| 2.0.0 | 2.0.0 |
| 2.0.0 | 2.0.1 |

15.6.4 Nebula Spark Connector

| Spark Connector version | Nebula Graph version |
|-------------------------|----------------------|
| 2.0.0 | 2.0.0 |
| 2.0.0 | 2.0.1 |

15.6.5 Nebula Flink Connector

| Flink Connector version | Nebula Graph version |
|-------------------------|----------------------|
| 2.0.0 | 2.0.0 |
| 2.0.0 | 2.0.1 |

Last update: May 20, 2021

15.7 Comments

15.7.1 Legacy version compatibility

- In Nebula Graph 1.0, four comment styles: `#`, `--`, `//`, `/* */`.
- In Nebula Graph 2.0, `--` represents an edge, and can not be used as comments.

15.7.2 Examples

```
nebula> # Do nothing this line
nebula> RETURN 1+1;      # This comment continues to the end of line
nebula> RETURN 1+1;      // This comment continues to the end of line
nebula> RETURN 1 /* This is an in-line comment */ + 1 == 2;
nebula> RETURN 11 +      \
/* Multiple-line comment      \
Use backslash as line break. \
*/ 12;
```

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

15.7.3 OpenCypher Compatibility

You must add a `\` at the end of every line, even in multi-line comments `/* */\`.

```
/* The openCypher style:
The following comment
spans more than
one line */
MATCH (n:label)
RETURN n
```

```
/* The native nGQL style:  \
The following comment \
spans more than      \
one line */          \
MATCH (n:tag)         \
RETURN n
```

Last update: May 25, 2021

15.8 Identifier Case Sensitivity

15.8.1 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;
[ERROR (-8)]: SpaceNotFound:
# my_space and MY_SPACE are two different spaces
```

15.8.2 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: May 20, 2021

15.9 Keywords and Reserved Words

Keywords 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. Non-reserved keywords are case-insensitive. To use reserved keywords as identifiers, quote them with back quotes such as `AND`.

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

// SPACE is an unreserved keyword.
nebula> CREATE TAG SPACE(name string);
Execution succeeded
```

TAG is a reserved keyword. To use TAG as an identifier, you must quote it with a backtick. SPACE is a non-reserved keyword. You can use SPACE as an identifier without quoting it.

Note

There is a small pitfall when you use the non-reserved keyword. Unquoted non-reserved keyword will be converted to **lower-case** words. For example, SPACE or Space will become space.

```
// TAG is a reserved keyword here.
nebula> CREATE TAG `TAG` (name string);
Execution succeeded
```

15.9.1 Reserved Words

The following list shows reserved words in nGQL.

```
ADD
ALTER
AND
AS
ASC
BALANCE
BOOL
BY
CASE
CHANGE
COMPACT
CREATE
DATE
DATETIME
DELETE
DESC
DESCRIBE
DISTINCT
DOUBLE
DOWNLOAD
DROP
EDGE
EDGES
EXISTS
EXPLAIN
FETCH
FIND
FIXED_STRING
FLOAT
FLUSH
FORMAT
FROM
GET
GO
GRANT
IF
IN
INDEX
INDEXES
INGEST
INSERT
INT
INT16
INT32
INT64
INT8
```

```

INTERSECT
IS
LIMIT
LOOKUP
MATCH
MINUS
NO
NOT
NULL
OF
OFFSET
ON
OR
ORDER
OVER
OVERWRITE
PROFILE
PROP
REBUILD
RECOVER
REMOVE
RETURN
REVERSELY
REVOKE
SET
SHOW
STEP
STEPS
STOP
STRING
SUBMIT
TAG
TAGS
TIME
TIMESTAMP
TO
UNION
UPDATE
UPSERT
UPTO
USE
VERTEX
WHEN
WHERE
WITH
XOR
YIELD

```

15.9.2 Non-Reserved Keywords

```

ACCOUNT
ADMIN
ALL
ANY
ATOMIC_EDGE
AUTO
AVG
BIDIRECT
BIT_AND
BIT_OR
BIT_XOR
BOTH
CHARSET
CLIENTS
COLLATE
COLLATION
COLLECT
COLLECT_SET
CONFIGS
CONTAINS
COUNT
COUNT_DISTINCT
DATA
DBA
DEFAULT
ELASTICSEARCH
ELSE
END
ENDS
FALSE
FORCE
FUZZY
GOD
GRAPH
GROUP
GROUPS
GUEST
HDFS
HOST
HOSTS
INTO

```

```
JOB
JOBS
LEADER
LISTENER
MAX
META
MIN
NOLOOP
NONE
OPTIONAL
OUT
PART
PARTITION_NUM
PARTS
PASSWORD
PATH
PLAN
PREFIX
REGEXP
REPLICA_FACTOR
RESET
ROLE
ROLES
SEARCH
SERVICE
SHORTEST
SIGN
SINGLE
SKIP
SNAPSHOT
SNAPSHOTS
SPACE
SPACES
STARTS
STATS
STATUS
STD
STORAGE
SUBGRAPH
SUM
TEXT
TEXT_SEARCH
THEN
TRUE
TTL_COL
TTL_DURATION
UNWIND
USER
USERS
UUID
VALUE
VALUES
VID_TYPE
WILDCARD
ZONE
ZONES
```

Last update: May 20, 2021



<https://docs.nebula-graph.io/master/master>